AD-A079 626 PATTERN ANALYSIS AND RECOGNITION CORP ROME N Y F/G 5/2
ADVANCED QUERY TECHNIQUES.(U)
OCT 79 C P MAH, J M MORRIS F30602-77-C-0161
UNCLASSIFIED RADC-TR-79-260 NL

1 OF 2
ADA
079626

12 LEVEL II

RADC-TR-79-260
Final Technical Report
October 1979

AD A079626

ADVANCED QUERY TECHNIQUES

Pattern Analysis & Recognition Corporation

Dr. Clinton P. Mah
Dr. John M. Morris

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441

80 1 21 057

This report has been reviewed by the RADC Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-79-260 has been reviewed and is approved for publication.

APPROVED:

ZBIGNIEW L. PANKOWICZ
Project Engineer

APPROVED:

HOWARD DAVIS
Technical Director
Intelligence & Reconnaissance Division

FOR THE COMMANDER:

JOHN P. HUSS
Acting Chief, Plans Office

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (IRDT), Griffiss AFB, NY 13441. This will assist us in maintaining a current mailing list.

Do not return this copy. Retain or destroy.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: RADC-TR-79-260	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle): ADVANCED QUERY TECHNIQUES	5. TYPE OF REPORT & PERIOD COVERED: Final Technical Report, 19 Sep 77 – 19 Jun 79
	6. PERFORMING ORG. REPORT NUMBER: N/A
7. AUTHOR(s): Dr. Clinton P. Mah; Dr. John M. Morris	8. CONTRACT OR GRANT NUMBER(s): F30602-77-C-0161
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Pattern Analysis and Recognition Corporation, 228 Liberty Plaza, Rome NY 13440	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 62702F, 45940119
11. CONTROLLING OFFICE NAME AND ADDRESS: Rome Air Development Center (IRDT), Griffiss AFB NY 13441	12. REPORT DATE: October 1979
	13. NUMBER OF PAGES: 182
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Same	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION DOWNGRADING SCHEDULE: N/A

(12) 184

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer: Zbigniew L. Pankowicz (IRDT)

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Intelligence Data Processing
Computational Linguistics
Natural Query Languages
Relational Data Models
Target Database Accessing

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The report describes an RADC sponsored R&D effort directed at providing an improved natural language access to differently formatted target databases. The end product consists of a testbed system designed for minimum dependence on any particular target database, hardware or operating system, and implemented for medium scale architecture. Section I defines functional characteristics of on-line intelligence information systems within the current state-of-the-art; describes the rationale of the AQT effort, and provides a comparison between the AQT approach and other approaches inherent in the (Cont'd)

DD FORM 1473, 1 JAN 73

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

390 101

Item 20 (Cont'd)

existing information systems with a practical orientation. Section II describes basic concepts of the AQT approach (extended relational data models, intermediate query language, table driven translation). Linguistic implementation of natural language query techniques is provided in Section III. Section IV deals with the methodology of accessing differently formatted target databases. Section V describes some special problems in querying target databases (e.g., generic keys, ellipsis, purging a context, conversational postulates). Section VI constitutes a detailed description of the presently available AQT testbed system. Section VII provides criteria for evaluation of user interface languages for database management systems. Section VIII is a statement of conclusions including present status and results; operational evaluation criteria; areas for further work, plans, summary and directions.

Abstract

The idea of relational models for data bases can be extended in a straightforward way to yield a fairly simple scheme for natural language access that can be easily tailored to any particular target data base of formatted records. A major part of this scheme, including a processor for queries in English, has been implemented in FORTRAN on a DEC PDP-11/70 as a demonstration question-answering system with Soviet aircraft data. The demonstration system suggests a design for a low-cost highly portable natural language access facility immediately applicable to existing intelligence data bases, either singly or several at the same time.

Contents

b. transparency

c. practical approach to understanding language

d. simplicity in design

e. facilitate experimental use

2. Directions

a. making natural language widely available

b. new information flows for intelligence

c. common access to multiple data bases

EVALUATION

The objective of this R&D effort consists in designing a next generation query facility that will supply deficiencies of information systems within the current SOTA, such as the requirement for special training and considerable experience; arbitrary and tedious access protocols; applicability to only one target database and requirement for knowledge of database structure/content; incompatibility with other information systems, and poor adaptability to new user needs.

The AQT approach offers the advantage of natural language access by non-expert computer users to differently formatted target databases. This design feature eliminates the requirement for special training, minimizes the dependence on one particular target database, and cancels the requirement for knowledge of database structure/content. Furthermore, the AQT approach imposes no restriction on access paths and provides a uniform access protocol for different target databases.

The AQT approach also provides portable technology applicable to medium scale architecture. The present testbed version is implemented in FORTRAN on a DEC PDP 11/70 under RSX 11/D, with restrictions minimizing dependence on this particular hardware/OS configuration. A fully operational advanced query facility will be compatible with Standard Software Base (SSB) including SARP data management system.

The subject effort has provided a valid design for a low cost, highly portable query facility with natural language access to the existing intelligence databases regardless of their formatting features. The facility will be capable of accessing either a single database, or several databases at the same time. The current follow-on effort consists in development of a prototype for hands-on experimentation; research into user needs and query language requirements; on-site or off-site application of the prototype to different intelligence databases on trial basis, and continuing evolution of the query facility through user feedback, ultimately leading to a general utility.

ZBIGNIEW L. PANKOWICZ
Project Engineer

SECTION 1

Introduction

1.1 On-line information systems for intelligence

Intelligence analysis is basically a process of sifting out and piecing together data so as to produce information pertinent to decision-making. Where uncooperative subjects are involved, this will seldom be as straightforward as looking at a single message, photograph or other collected data item. More typically, an analyst will have to work inferentially from many incomplete and possibly even conflicting data items. In such a situation, it would seem desirable to have as much data as possible, but in practice, the ability to collect data far outstrips the ability of any unsupported analyst to make sense of the data. Accordingly, many types of online information systems have been developed to allow computers to help in the management of large files of intelligence data.

With mass storage devices and control processing units

continuing to improve in speed and capacity while declining in cost, the technical and economic feasibility of online information systems no longer are in doubt. Making this information accessible by users with actual need of it, however, remains a problem because these users typically lack training as computer programmers. Simply having data stored on line does automatically insure that it can be used effectively. If fact, there is often a problem with too much data if access to it is gained only through learning many different access procedures.

As a means of making online information more conveniently available to non-expert computer users, two key concepts have emerged as major areas of ongoing research and development: the relational modeling of data bases and allowing requests for information to be expressed in a natural language like English.

- The technique of relational modeling allows an information user to approach a data base in terms of the semantic dependence between data items without consideration of how this is actually implemented in the data base. The user sees data as a set of abstract relations defined over primitive data types; for example, the relation EMPLOYEE holding for the data types NAME, ID#, JOB CATEGORY, and AGE, typically written out as EMPLOYEE(NAME,ID#,JOB CATEGORY,AGE). The user is

allowed to manipulate the various items of data actually associated with a given relation without having to know such technical details as the physical device on which they are stored, the logical organization of storage on the device, and the access method for getting to the data. All of this would be handled by system software and made transparent to a user.

o A natural language capability allows an information user to refer to the contents of a data base in the same way that a person would talk about them in a language like English. This goes further than simply including English keywords in a formal query language or having the syntax of that language imitate the form of English sentences; naturalness implies that a person can also use the various familiar semantic conventions of a language like English to describe the various kinds of logical dependence within a data base. A person should be able to use a natural query language without having to know anything more than the general fact that a given data base contains certain kinds of information.

Both relational modeling and natural language querying have been implemented in a wide variety of experimental systems, often

with the two in combination. The success of these systems in simplifying data base access for non-expert users has stimulated interest, but it has proven difficult to introduce them into actual working environments. The problem is both technical and managerial. On the one hand, small experimental systems tend to scale up poorly, since they tend to bog down computationally as data bases approach the size of those typically in the real world; on the other hand, information system managers are reluctant to commit themselves to a new, fairly complex system that they cannot easily evaluate.

Trying out a relational or natural language query system can be costly. It may involve a great deal of special programming as well as translation of data from existing formats into formats acceptable to the system. The effort will also be risky in that there is no guarantee that the system will meet the perceived needs of a given user or that it will even run with a given data base. There may also be problems of integrity if multiple copies of data have to be made because of incompatibility between the requirements of natural language query processing and existing data processing.

The Advanced Query Techniques (AQT) project was initiated in an effort to smooth out some of the difficulties in getting the technology of relational data bases and natural language to users

in need of convenient and flexible access to online intelligence data bases. The goal of the project was first to identify a subset of such technology appropriate to intelligence data analysis and then to build a system to demonstrate the effectiveness of this technology to potential users intelligence. To accomplish this, we have developed an extended kind of relational data model designed specifically with natural language in mind and have implemented a demonstration system on a DEC PDP-11/70 using the extended data model as a means of translating natural language queries into references to a target data base.

Some examples of queries handled by the ADT demonstration system:

What is the fuselage length of the Foxbat?
Of the Flogger?
Average wingspan of all MIGs?
How many fighters or interceptors carry the
Atoll missile?
What configurations?
Give me their combat radius.
Their gross weight.
Which of these have 2 crewmen?
1 crewman?

The record of an actual session with the demonstration system is given in Appendix A.

The demonstration system is written in FORTRAN and currently runs on either RSX-11M or -11D. It is set up to be table-driven so as to make it independent of any given data base. It will be the basis here for the design of a full prototype portable natural language query facility runnable on a processor as small as a DEC PDP-11/45. This facility will make it possible for practically any user to have natural language access to existing formatted data bases with no additional hardware and with no additional software except possibly for special routines to handle unusual data types or linkages.

1.2 Background

Work on AQT began as an outgrowth of an evaluation of the REL (Rapidly Extensible Language) system [10] by PAR Corporation for RADC (contract #F30602-75-0241). Although REL was and still is a promising natural language data analysis system, we found it most interesting in the areas of further research that it suggested. One particularly striking problem was that of how to name the binary relational data structures employed by REL so as to make them as transparent as possible to a user. The REL

system never directly addressed this issue other than to provide a way for users to define local synonyms for existing names.

In a broader context, the naming problem turns out to be only one aspect of a more general difficulty that relational data systems tend to gloss over. It is often implied that a standard relational representation of data such as third-normal form is in some sense natural and that its organization and labeling should therefore be obvious to any user. This is not necessarily so, however; for as we shall show, there are many different possible realizations for a given item of data in a given canonical form of relational representation. To make the organization of a relational data base completely transparent to a user, we have to make the choice of relational structures transparent as well.

To attack this problem, we started by extending the notion of relational data structures to include a hierarchical ordering of relations and to allow relations to inherit key fields from further up in the hierarchy. With strict rules on the naming of relations in this framework, the model turns out to be quite transparent to a user and in fact suggests how English queries might be interpreted within the model. The model is still sufficiently simple in structure, however, that mapping it back onto a target data base can be accomplished through a straightforward table-driven procedure.

The use of an intermediate logical model in natural language data base access has the advantage of splitting the problem into two separate subproblems: natural language processing and data base access processing. The natural language part unexpectedly turned out to be the easier one, largely because of the availability of the PARLEZ [8] system for developing query language grammars and parsers driven by them. We were fairly quickly able to produce a demonstration on a DEC PDP-11/70 for the translation of a significant subset of English into an intermediate query language directed at a given relational hierarchy serving as a logical data model. Handling target data base access took a great deal more time.

At first, our plan was to use existing software from the SARP [16] data management facility to handle data base activities, since this would reduce the amount of programming required to bring up an AQT demonstrate system. Delays in the delivery of SARP and inadequate documentation, however, forced a change of course. Given the imperative of showing the effectiveness of AQT, we decided to put SARP aside temporarily and to bring up a simple data base ourselves on a PDP 11/70. To avoid being unrealistic, we placed no prior conditions on the organization of the data base and designed the data access part of the demonstration system to avoid any assumptions about the structure of the target data base. This in effect made the

demonstration system into a preliminary version of a full AQT facility.

In this capacity, the demonstration system was a valuable testbed for developing techniques for specifying data base structure in table form, generating data access paths, controlling data base searches, resolving queries dependent in meaning on a preceding query, selecting and formatting output, and computing various statistical functions of data. The demonstration system by running on a PDP-11/70 also shows the feasibility of implementing an AQT facility even on medium-scale machine architecture.

1.3 Other Approaches

Programming a machine to understand natural language is always enormously more difficult than most people expect. It was soon after the construction of the first electronic computer, for example, that the first proposals for automatic translation of foreign languages were made. At the time, this seemed to be within reach of the unprecedented computational power becoming available; but raw power by itself turned out to be insufficient. With steady technological progress, there are now larger and faster processors, more sophisticated programming

languages and methodologies, and a deeper theoretical grasp of natural language; but even these have failed to bring about the "talking computer" of popular myth.

The major difficulty in building natural language systems is in limiting the problem to make it manageable. In human beings, linguistic behavior is closely intertwined with intellectual, emotional, and social behavior so that any attempt emulate human linguistic behavior soon gets into complications. The problem of getting computers to read simple stories illustrates the point here; there is not only a tremendously rich array of language to be dealt with somehow, but also the formidable task of incorporating a broad supportive repertory of linguistic and non-linguistic behavior in a system because any real understanding of a story requires being able to make an appropriate response to it.

In this context of the natural language problem, we can distinguish two basic approaches to building systems. The first is the longstanding "talking computer" approach that envisions making computers full partners of human beings by ultimately giving them the power of speech. The focus here is typically on attacking difficult problems in natural language understanding with the goal of developing new techniques for handling them. Systems built along these lines are often highly successful (c.f.

Winograd's SHRDLU system [14]), but the measure of success for such a system tends to highly correlated with the number of new difficult problems that it opens up.

An altogether different way to build natural language systems is what we might call the "nuts and bolts" approach. Instead of attempting to come up with major breakthroughs, we can take the wide range of language processing techniques available and try to fit them together in order to solve a practical problem. In the last several, this approach has become especially prominent with an orientation toward improving the accessibility of information stored in on-line computer data bases. A query facility based on AQI falls into this category, and so to put it into perspective, we need to look at the other systems around.

1.3.1

1.3.1.1 -

The REL system has been mentioned already [12]. This is a relational data analysis system with an English query interface that is readily extendable by a user while at a terminal. The extensions are mostly like macro definitions, serving to expand a

particular user's way of saying things into concepts and operations already known to the system. These kinds of extension would allow a user eventually to tailor a query language to be maximally convenient for a given application.

REL was originally implemented on an IBM 360/65 in assembly language for speed and compactness. Current versions are being directed toward other machines, including one special-purpose minicomputer built to run REL. The system is designed for I/O efficiency, relying on data being stored in a special binary relational format and on direct physical access to secondary storage, possibly having to go around the file system management for a processor. Language processing is done through a syntax-driven scheme, using a parser designed to handle the most general rewrite grammars.

On the whole, REL appears to work out as a practical system aimed at data analysis as carried out in the real world. Actual non-expert users seem able to apply the REL definitional facility in effect to write data processing programs in English. The major shortcoming of REL is probably its restrictive data formatting requirements; data physically has to be stored in a specific way before REL can work with it. This also leads to difficulties where fairly complex data is involved; although binary relations are theoretically sufficient to represent any

kind of data structure, it is not clear that the REL language processing capability can make arbitrarily composed binary relations easily accessible to a user. Finally, there is a problem with the naming of relations and fields that prevents the system from being fully transparent (see Section 2.1.).

1.3.1.2 -

ROBOT [5] is a commercially available system intended primarily for management information applications. Its basic idea is simple, but effective - namely that a data base itself can be used as an extension of a dictionary, practically eliminating any need for a separate world model to guide language analysis. In practice, this involves fully inverting a data base to get index lists of the records containing a given value in a given field. The system is designed to take advantage of the index lists wherever possible in data base search, reducing the need to read in target data base records.

Currently, ROBOT is sufficiently well established to have an organized users group. Non-technical managers and their staffs appear to find the system easy to work with and the information returned by the system useful. The system is fairly modest, running on an IBM 370/155 with 100K to 200K bytes of memory. The

natural language processing is done with an augmented transition net (ATN) parsing scheme [15].

It is unclear yet how complex a data base ROBOT can handle effectivley. The problem is the requirement fo full inversion, which may not always be reasonable to do. The ROBOT system at present seems restricted to extremely simple data file organizations, mostly linear files of records that would typically be searched with a sequential access method. This is probably adequate for some business applications, but it may be too simplistic for a highly structured S&T data base.

1.3.1.3 -

Planes is a query facility that translate natural language queries into formal queries directed at a relational data base containing maintenance information about naval aircraft. It takes advantage of the restriction of its domain of discourse to simplify the overall problem of natural language processing. The system aims not at emulating human language comprehension, but at engineering a practical system for a particular information problem.

The system is designed to be tolerant about the form of

queries. It avoids being strict about grammaticalness by using a local parsing scheme based on augmented transition nets to get a set of phrases from a query and then using conceptual case frames to tie these together. It also keeps close track of the context of queries so as to be able to fill in parts left incomplete. The language processing capability of PLANES on the whole is fairly sophisticated relative to the means available for it.

At present, PLANES is not intended for use beyond an aircraft maintenance data base, although its use of a relational data interface allows data base independence to some extent. The system specifically requires all data to be stored in a non-hierarchical tabular format, however. PLANES is implemented on a DEC System 10 in LISP.

1.3.1.4 -

LADDER (6) is a general data base access system with a natural language interface defined with LIFER, a special natural language processing support package. The system consists of three major components: a query acceptor that produces formal retrieval specifications from natural language input, a translator that produces queries in a given target data base management system access language from specifications, and an

access manager that finds the location of files referenced by queries and goes through all the necessary procedures for getting access to them. Files in a target data base may be scattered across several different computer systems linked into a network.

The natural language interface is designed to be extendable by a user submitting examples of new query forms and specifying how they are to be interpreted relative to recognized forms. The entire interface in fact can be tailored interactively to a given application, there being no initial core grammar supplied. The only restriction to defining a query language from scratch is that the person specifying it has to be familiar with the LISP programming language, since the basic system is implemented in INTERLISP on a DEC KL-10. The facilities simply for defining new terms and paraphrases are much easier to use, though.

LIFER is independent of a target data base to the extent that its translator and access manager components can be replaced. These modules appear to contain target data dependent code, although most of this is probably connected with particular data base management systems and not with the contents of a data base. The translator module especially would have to be changed if the target data base management system were changed.

1.3.2

The AQT approach has something in common with each of the systems mentioned here, but most closely resembles LADDER. Like LADDER, an AQT facility is based on a multi-stage translation process that attempts to make the structure of a target data base transparent to a user through methods that can be applied to data bases of different structure and content. The orientation of AQT, though, is slightly different in its employment of a kind of relational data model and a basic query language grammar, bring it closer in this respect to REL and PLANES.

As natural language systems, all four of the systems here as well as AQT appear to aim at about the same practical level of competence, adequate for their intended use with data bases but not spectacular. Although the techniques vary, all the systems seem to agree on the importance of problems like ellipsis and simple reference. Where the other systems diverge most from AQT is in the conception of an AQT facility as a small, low-risk system running with limited resources in a user's operating environment; this is reflected in the implementation of the AQT demonstration system in FORTRAN on a DEC PDP-11/70.

SECTION 2

Concepts

2.1 Extending relational data models

The central concept underlying the AOT facility is an extension of the notion of relational data structures [2]. Relational models for data bases improve a great deal on classical approaches by making access methods and data record organization transparent to a user; but they still require the user to know something about the structure of a data base. The problem is that there is no one natural way to express any given data in terms of relations. For example, data about the hardening of support airfields against airstrikes might conceivably be organized in many different ways:

```
AIRFIELD        [..., type=support, hard=yes]

SUPPORT AIRFIELD[..., status=hardened]
```

FIXED TARGET [..., class=support airfield, vulnerability=hard]

.

.

.

where relation names appear in upper case to the left of a set of brackets and field names and associated values appear within brackets. As can be seen, what constitutes relation names, field names, and value designations can be highly relative; variations in relational structures can be quite radical, going far beyond the standard algebraic operations used to derive new relations from old ones.

To get at information in a relation data base, a user would ordinarily have to know about its particular logical organization: the actual names of defined relations, the actual names of fields contained in each relation, and the types of values associated with each field, as well as the pragmatic significance of certain values being associated with certain fields in certain relations. Because these are all somewhat

arbitrary, however, they are inessential to the problem of data base access, serving only to complicate the situation for a given user. If a data base is known to contain information about the hardening of support airfields against airstrikes, then a user ought to be able simply to ask "Is airfield XXX hardened?" without having to worry about the various additional details imposed by a given relational representation.

The difficulty with the many different ways that relations, fields and values might be defined for any given data base turns out to be a key clue to solving the problem, though. If the variants of a relational representation are viewed from a linguistic standpoint, then we can interpret them as the result of the various kinds of syntactic transformations familiar to natural language [3]. The straightforward application of natural language analysis to relational nomenclature can therefore go far to make relational data structures more transparent to users. This is much more, it should be noted, than simply allowing the queries to a relational data base system to imitate English syntax.

An obvious starting point for studying transformations and data base nomenclature is to look at how words are combined to form the names of relations and fields in general. In the context of a given data base, we tend to see that certain label

words (nouns) seem especially important, serving as the principal word of descriptive relation names or as distinguishing qualifiers in both relation and field names. These are in a sense invariant under the various linguistic transformations on relational data structures; for example, a transformation may involve removing such a label word from a relation name, but only to put it elsewhere, perhaps as a new modifier of field names in the relation.

```
FUSELAGE DIMENSION        [..., length=xxx, width=yyy]

          becomes

DIMENSION                 [..., fuselage length=xxx,
                                fuselage width=yyy]
```

.

.

.

In natural language discourse, the important label words for

a given data base tend to have a definite partial ordering in terms of their relative position in different possible relation and field names: whenever two such label words appear in a name, one almost always precedes the other. For example,

AIRCRAFT WING DIMENSION
WING DIMENSION
AIRCRAFT DIMENSION
AIRCRAFT WING

but not,

WING AIRCRAFT DIMENSION
DIMENSION WING

(There is, to be sure, a habit in some branches of the military to construct noun phrases with backwards modification, but this is unnatural with respect to common usage.) The partial ordering here is simply a generalization of patterns of speech and yields a natural hierarchy of label words for a given data base. This hierarchy defines a partial semantic analysis of the descriptive relation names for the data base and thus also an analysis of the data relations themselves.

A hierarchical decomposition of relation names provides the

basis for a simple extension of relational data structures. First, we will allow multiple occurrences of label words in a hierarchy so that it can become a tree (or a forest of trees). A data relation now descriptively corresponds to a downward path of label words in the hierarchy. We can reinforce this correspondence by mapping over the field names of data relations now as well, moving them up in the hierarchy until they are semantically dependent on the concept denoted by a label word at some level. Within a relational hierarchy, field names with a common label word modifier can be analyzed further by creating a new sublevel for that label word and moving these names less the modifier down to that sublevel. This procedure results in a new set of data relations, each named by a single label word and having a hierarchical ordering.

```
AIRCRAFT [nato name=xxx, ...]
:
: . .FUSELAGE [...]
:        :
:        : . .DIMENSION [length=yyy, ...]
:
: . .WING [type=tttt, ...]
         :
         : . .DIMENSION [dihedral angle=zzz]
```

This sort of structure will be called a "relational hierarchy"; it could be more formally defined as a partially-ordered set of relational data structures with single-word labels, where the

hierarchy models typical usage of words in simple English noun phrases when talking about a given data base.

To obtain a relation hierarchy as a logical model for a given data base, we first get a standard relational model for that data base with relations defined so that all non-key fields of any relational tuple are functionally dependent on each of the key fields of that tuple; this is known as "third-normal form" and can always be derived for a data base. Having a relational model, we can next derive a relational hierarchy from the relation names as above, substituting synonyms where necessary to allow as much merging as possible of downward paths in the hierarchy. In this way, we get in effect a maximally factored third-normal form relational model.

As a kind of logical data base model, a relational hierarchy has the advantage of being practically invisible to a user. Its structure is simply an analog of natural language word modification and should not have to be spelled out for a user who knows English and who is a little familiar with the contents of a given data base. What we have done here is to eliminate some of the degrees of freedom in a relational approach so as to make it more compatible with the way that non-expert data base users might intuitively perceive information. The scheme is extremely simple, but in practice, it seems to work out fairly well as a

basis for computer programs to make sense of English queries directed against a formatted data base.

Given relational hierarchies as a means of query interpretation, a natural language query facility falls neatly into two distinct parts: a language analysis component that parses incoming queries and maps them into data references relative to a relational hierarchy data model; and a target translator component that maps references to a logical data base model into references to an ultimate target data base. Having a relational hierarchy as an intermediate step permits language analysis to be developed independent of the target translator, and thus independent of the target data base itself. This greatly enhances portability of the query facility at the front end.

2.2 An intermediate query language

Translating an input query into an intermediate language is advantageous in several ways. The most important consideration is modularity: it breaks up the query processing problem into two more manageable pieces from the standpoint of established computational techniques for compiling and translating languages and for retrieving data through formal queries. This sort of

modularity also enhances the ultimate portability of a natural language query facility in that the input query processor can be designed with a minimum of assumptions about an actual target data base.

From the perspective of the user, the display of an intermediate translation is helpful for verifying that query processing is in fact proceeding correctly. The derivation of relational hierarchies on the basis of linguistic modification makes it fairly easy for a user to understand an intermediate query without really having to know all the things necessary in order to construct one correctly. There is of course a hazard in exposing a user to some of the technical details of query processing, but on the whole, it seems more balanced off by making query processing less mysterious to the user and giving the user some control over it.

An altogether different motivation for intermediate translation arises because of the need to deal with the problem of pronouns and other nominal expressions that have to be understood in context. In general, this requires that an access facility be able to save previous queries in a way that allows information to be extracted from them when needed to interpret references of a current query. The intermediate language form of queries turns out to be convenient for this purpose (see Section

4.1 for details).

We might draw an analogy with the similar technique of having an intermediate language phase in a compiler for a programming language. It is a standard approach for making code optimization easier to do and for designing compilers to be portable; and it can be effective as well in other language applications. By being careful in the choice of an intermediate language, we can greatly simplify the problem of processing any language, something especially important when dealing with linguistic complications on the order of those arising in English.

The intermediate query language assumes that all data of interest can be organized into a logical hierarchy of relations, where a relation could be thought of as an abstract record type with certain defined fields for holding specific kinds of data. For example,

```
AIRCRAFT [country, designation]
:
:  .  .FUSELAGE
:  :
:  :  .  .DIMENSION [length, width]
:
:  .  . ENGINE [#, manufacturer, thrust]
```

The logical hierarchy of an aircraft data base with 4 relations and 7 fields defined.

An intermediate query is a series of "clauses" referring to such a hierarchy. It is expressed as a list of the clauses terminated by the symbol (.)

CLAUSE - 1

. CLAUSE - 2

.

.

.

. CLAUSE - n

(.)

A "." preceding a clause indicates dependence on a previous clause; it is a kind of continuation marker. The first clause in an intermediate query may or may not be dependent; all subsequent clauses in the query must be.

Each clause traces a path through the relational hierarchy of a data base, starting from the top of a hierarchy for independent clauses and branching off previously defined paths for dependent clauses. Each path can be thought of as corresponding to a chain of relational instances (actual logical data records) matching the conditions of a clause.

A clause is expressed in the general form

R1 R2 ... Rn(Mn)[Q1,Q2,...,Qk]

where

Ri are relation names in a data hierarchy

Mi are special action markers

Qi are predicate qualifiers on relational fields (enclosed in square brackets)

Relation names will always be one word long to avoid ambiguity problems. The sequence of relations R1 R2 ... Rn in a clause must trace a continuous downward chain in a relationl hierarchy. For an independent clause, R1 must be the top of the hierarchy. In the relational hierarchy from above,

". FUSELAGE DIMENSION"

"AIRCRAFT ENGINE"

"AIRCRAFT FUSELAGE DIMENSION"

"AIRCRAFT" are all possible legal sequences.

A qualifier Qj consists of a list of field specifications separated by commas. each specification is of the form

Fieldname arithmetic-comparison-operator value

There are a number of conventions with field names:

- o a field name in an intermediate query may have a function name associated with it, with the function name enclosed in parentheses. For example, [(maximum) length=*]. In this case, the procedure is first to look for a field name "maximum length"; if there is no such field in a data base, then the field "length" is searched for in the data base model and if found, the function "maximum" is applied to all fields of relational instances to yield a resultant value for comparison or return.
- o secondary keys enclosed in <<...>> may be part of a field name in an intermediate query. These may be in two forms, <<X=Y>> or simply <<X>>. These allow for the value of a field to be actually an array. The first form <<X=Y>> selects a column or row or plane of an array explicitly. The second form allows for field names of the form "X field name";

if no such field exists, then field is assumed to be an array and X is assumed to be a parameter for selecting a value associatively. It is assumed that given X, we can find what parameter it must specify by looking at the array. In other words, the intermediate query need not say this explicitly.

o "#" is a special field name for how many of an object corresponding to a relation is associated with an object of the parent relation.

Values will have associated conventions also. It is assumed that default units of measurement are associated with fields of a data base and that these apply if omitted from a value specification. If an explicit unit is specified, then the intermediate query language processor has the responsibility of converting units in making comparisons. Exact matches of numerical values are not mandatory; the data base description should specify what the range of exactness should be for various fields. (This is not the same as a measurement error.)

For example,

designation=MIG-25, country=UR

length>100 feet

A value may be the symbol *, which is the wildcard. This will match any value of a field provided that the field is actually defined for a given relation instance. A marker Mi can be a uniqueness indicator (!) or a question indicator (?), (#?), (Y/N?) or both. Question indicators, however, are mutually exclusive. For example,

(?)(!)

(#?)

Markers Mi and qualifiers Qj are optional in a clause. If a qualifier is null, then it matches anything providing that the corresponding relational instance is actually defined.

For an independent clause, the procedure for interpretation is as follows for qualification:

- o start at the top of the hierarchy.
- o retrieve all relational instances compatible with qualifiers.
- o if a lower level of relations is specified, retrieve all lower relational instances chained to those already retrieved and having compatibility with qualifiers at the lower level.
- o continue this until either the end of the clause is reached or no relational instances satisfy a qualifier.

For dependent clauses, the procedure is similar:

- o start at the current place in the relational hierarchy, i.e. where the preceding clause stopped.
- o try to interpret the clause as applying to the subhierarchy of relations at the current place and referring only to relational instances satisfying the preceding clause.
- o if the first relation of the dependent clause does not exist immediately below the current relation, back up one level in the relational hierarchy and try again.
- o in backing up, retain only those higher-level relational instances chaining with a matched relational instance at the lower level.

The procedure for qualification selects certain subsets of relational instances for each of the relational types specified in a query. The markers of an intermediate query identify what subsets are of interest, as well as the type of action to take with them in response to the query.

- o (n!) This specifies that only n relational instances should match. If n is preceded by the operator '>' or '<', then it specifies that more than n or less than n should match. If n is omitted entirely, the marker is for definitions with a specific number.

- (?) This specifies that all key fields and all * fields of the marked relation are to be presented to the user for the matched relational instances.
- (#?) This asks only for a count of relational instances matched.
- (y/n?) This asks for a yes or no answer, depending on whether any relation instances match the conditions of a query.

For the most part, the conventions here should be adequate as a basis for question-answering applications. The only further detail that needs elaboration is the problem of reference across queries, where the first clause of a query is dependent. This would work in much the same way as reference within a query, but with a complication.

The problem with cross-query reference is that this does not always refer to relational instances identified in a previous query. Often the intent is that the current query incorporate entire clauses from a previous query so as to identify completely different relational instances (i.e. anaphoric reference). The procedure for dealing with this will be as follows:

- Assume the preceding intermediate query is available;

merge the current one into it, with the current one superseding if there are conflicting differences.

- If the result is the same as the preceding intermediate query except for additional field specifications, then interpretation of current query can proceed by applying those specifications to the relational instances matched by the preceding query.
- If a field specification is changed without any changes in question indicators or addition of clauses, then interpretation must start over from the beginning with the merged query. (This situation is anaphoric).
- If a question indicator is changed without the introduction of any new indicators, then the interpretation will proceed on the basis of other changes in the merged query, if any.
- If a new clause with a question indicator is introduced by the current query, then all unrepeated clauses from the preceding query having a question indicator and tracing a divergent path in the hierarchical relational hierarchy must be deleted, and query interpretation must start over again. All other previous question indicators will be dropped in any event.
- If no new clause contains a question indicator and there is no previous question indicator at a hierarchical level above a new clause, then interpretation can proceed

directly from the results of the preceding intermediate query.

- o If there is a new clause without a question indicator below a previous question indicator combined with a definiteness indicator, then interpretation will proceed as above as if there were no new clause. If the definiteness indicator is absent, however, then this is deemed ambiguous. The requester is notified of this, and it is assumed that interpretation should start over from the beginning with the full merged query.
- o The default for interpretation with reference is always to proceed from the results of the previous query.

Examples:

A sequence of queries with reference

```
AIRCRAFT(?)                   "How many planes
. ENGINE[#=4]                 have 4 engines?"
(.)

AIRCRAFT(?)[country=ur]      "How many Russian
(.)                           ones?"
```

Here we reinterpret from "AIRCRAFT" level with the relation instances matched there.

. AIRCRAFT FUSELAGE DIMENSION(?)[length=30]   "How many
(.)   have length
                                                                        30?"

Ambiguity = length of 4-engine planes or length of 4-engine Russian planes?

Here we would assume the latter and would inform the user of that fact.

2.3 Table-driven translation

Because the AQT project was directed toward the development of general data base access techniques, we had to avoid any approach that depended on any particular kind of target data structure. This became especially important in implementing a demonstration system for AQT, for it had to be clear that its capabilities would apply to large, complex data bases from the real world as well as to a small, contrived demonstration data base. We therefore adopted a table-driven processing strategy for AQT, where data base dependent details of access would be kept separate from the operation of a system in external tables.

The syntactic analysis and rewriting of input queries were easy to handle in this manner because they involved the kinds of techniques already evolved for syntax-directed translation of

programming languages [1] and the construction of compiler-compilers. In the AQT approach, an input query language is defined by a table of rules of grammar and a dictionary table of vocabulary for a particular target data base. The semantics of a query language is defined by procedures associated with each rule of grammar and each dictionary entry; these procedures, encoded as strings to be read by special interpreter, are also kept external to a system.

For the translation of natural language queries into intermediate queries, the problem of AQT was to provide convenient but yet powerful enough facilities to define the kinds of grammars and dictionaries required for natural language and also to provide a flexible parser that could be driven by these grammars and dictionaries. Our solution to grammars was to express them using context free rules with extensions in the direction of van Wijngaarden grammars [13] for additional descriptive powers; these extensions were constrained so that we could still basically employ familiar techniques for parsing context-free languages. Dictionaries posed no major difficulty with a semantic scheme of string manipulation through text editing primitives; words could be defined simply as sets of ASCII character strings passed as arguments to semantic procedures. Section 3 will describe this is more detail and will go into an approach to a basic grammar for natural language

queries.

The AQT parser plus an interpreter for semantic procedures constitutes what we call the "AQT intermediate translator", producing formal intermediate query strings from natural language queries. To complete a query facility, we need a way of mapping an intermediate query into actual references to a target data base. One possibility is to translate it into a data access language already defined for the target base, but this is not always feasible; a data access language may not always exist or it may not be able to support the capabilities that we would want for natural language access, or it may be too hard to map into, requiring the equivalent of being able to generate programs automatically.

The approach currently taken by AQT is to access a target data base directly, assuming that it is organized into records or some similar data construct and that there exists a procedure for getting these records from the data base. The AQT target data base translator maps an intermediate query into a data access sequence of references to particular fields in particular records. The data access sequence is applied in searching a target data base to retrieve records matching specified conditions, and then selected fields of these records are printed in response to the original query.

Like the intermediate translator, the target data base translator can be organized as a table-driven scheme. It is a somewhat more difficult problem in that there are fewer established techniques to draw upon, but from any practical standpoint, it is unavoidable. The issue here is not only that of making it easier to bring up a query facility on different target data bases, but also that of accommodating the almost inevitable growth of a target data base when it is discovered to contain useful information accessible in a convenient way.

The basic tables involved in target data base translation define a correspondence between fields within a relational hierarchy and fields of target data base records and between functional dependence of fields in the hierarchy and data linkages implemented in the target data base. We cannot actually describe all possible correspondences of this sort because the ways of organizing a target data base are infinite; but we can accommodate the most common kinds of target data fields and linkages while allowing for the inclusion in a system of special procedures to handle the exotic cases. Section 4 and 5 go into this in more detail.

SECTION 3

Handling Natural Language Queries

3.1 Elements of a query language

We have to be more specific here about what we mean by a natural language, for the word "natural" in the past has been applied in a bewildering variety of senses. For example, COBOL is sometimes called natural because it employs English words extensively, REL [10] because it allows users to extend a language, SHRDLU [14] because it can make intelligent inferences, and most systems typically because they accept input with a syntax based on the structure of sentences in a language like English. In the extreme case, naturalness could conceivably be stretched to encompass anything not artificial, not associated with machines.

In AQT, naturalness has a restricted sense, largely because we are specifically addressing the problem of data base access. With any given data base, the kinds of responses that can be made

to a query are actually quite limited: answering yes or no to the existence of data items, giving counts of items, printing out some combination of fields from selected target data base records, printing out documentation about the data base itself, prompting the user where a decision cannot be made automatically, and displaying diagnostic information about the course of processing a query. The problem of natural language here therefore reduces to the question of how to let a person elicit such responses in a natural way.

From a practical standpoint, we can assume that a user of a query facility will generally want responses that yield information from a data base. This means that queries can be expected to refer to the structure and content of a data base as perceived by the user. Relational hierarchies are introduced here in an attempt to model a typical user's perception of data bases and therefore provide a way of classifying the possible referential constituents of queries, which in our case would include relations, field names, and literal values.

To make the scheme here as transparent as possible to the user, we do not require the user to be aware of referring to a relational hierarchy, and we do not of course put any restriction on how such (implicit) references to a hierarchy are combined in a query. We will assume only that queries will be such that they

are intelligible at least to the user entering them and that, in the absence of any other conventions for communication, the user will use something resembling English syntax.

Our approach to the analysis of queries will first be to identify individual words in a query in terms of semantic significance in a relational hierarchy: relation, field name, literal value, and so forth; this will be done through a dictionary of query language vocabulary. Then we will attempt to associate literal values with appropriate field names, field names with relations, and relations with other relations according to their hierarchical ordering; this will be done through a grammar of expected patterns of words with various kinds of semantic significance. The result of all this will be an intermediate query to be passed on for further processing.

3.1.1

To define a query language, we first of all need a grammar to specify the various basic constituents of the language and how these combine to form larger constituents up to the level of sentences. In AQT, a grammar consists of context-free rules (see [1]) of the form X->Y Z, stating that adjacent Y- and Z-constituents together can become an X-constituent regardless of

context, and X->Z, stating that a Z-constituent is also an X-constituent regardless of context. More complex rules of grammar than this are usually employed in natural language applications since context-free rules cannot fully describe the syntax of these languages, but for AQT, such rules simplify greatly the problem of parsing queries and with a few simple extensions can be quite adequate for query language description.

A serious descriptive shortcoming of context-free grammars is in the number of rules needed to deal with any nontrivial aspect of natural language. A simple example here is the case of noun phrases and determiners like "a" and "the." When a noun phrase already starts with a determiner, there are syntactic consequences like ruling out another determiner in front and so the noun phrase ought to be marked. This can be done by having a new constituent type DNP for determined noun phrases as opposed to just NP; but this also leads to a problem because DNP in many situations is still equivalent to NP, forcing us to duplicate many rules of grammar that replace NP with DNP. The problem worsens with more dimensions for marking a constituent type, because the number of extra rules needed grows exponentially.

Fortunately, there are ways around the difficulty here, and in fact grammars in AQT employ several of them to provide as much flexibility as possible in language specification. A simple, but

powerful enhancement to context-free rules is to allow syntactic features to be attached to a constituent type X as a qualifier, written as X[F,G,...,K] where F,G,...K are features. On the right side of a rule of grammar, features specify conditions on subconstituents that must hold for the rule to be applicable; on the left, the features are those that are to be turned on upon getting a new constituent through a rule. The attachment of features to syntactic categories allows a single rule of grammar to stand for an entire family of ordinary context-free rules; this kind of extension turns out to be along the lines of van Wijngaarden grammars, which have been shown to be adequate for describing any language recognizable by an automaton [13].

Another enhancement, in sense even more powerful, is incorporated in the semantics for a grammar. In AQT, the semantics of a rule of grammar is defined by a procedure attached to the rule, as done in the syntax-driven translation of computer programming languages [1]. The particular scheme in AQT, derived from the LINGOL system [9], allows semantic procedures to communicate through shared variables. The effect of this is about the same as for syntactic features in that a procedure for a rule can execute code conditionally on these variables and therefore make the rule equivalent to a family of rules with different semantics. A broader discussion of semantic procedures for rules of grammar comes in Section 3.2.

3.1.2

The AQT approach is flexible enough to let users define their own query languages from scratch by supplying grammars and dictionaries. In practice, however, this involves a great deal of work, much of it duplicated in different language since in AQT they all will first be translated into a fixed intermediate language. It is therefore reasonable and helpful for a query facility to include a basic query language grammar at least, something that could be elaborated on if necessary.

The basic AQT query language is a subset of English revolving mainly around nouns and noun modifiers, given our focus on data bases about things rather than events or processes. Verbs occur in the language, but play a fairly minor role since they seldom will map into anything specific in a relational hierarchy. Omission of verbs in a query is in fact allowed by AQT so as to reduce the work necessary in entering a query.

The basic AQT query language grammar mostly deals with words denoting relations, fields, and literal values in a relational hierarchy and how they combine within noun phrases. The purpose of the grammar was not to be exhaustive, but rather to provide a simple, practical scheme adequate for what might reasonably be expected in the actual querying of data bases. The scheme

essentially notes that, in a noun phrase, value references associate with field specifications and field specifications with relations, eventually amounting to a clause in an intermediate query. The systax for this kind of association in English is quite varied.

	relation	field	value
relation	"aircraft wing"	"aircraft name"	"wing swept"
field	"length dimension"	---	"type X"
value	"fighter aircraft"	"124 feet long"	---

This suggests that a basic query language grammar should be something simple like this:

```
SPECIFICATION->VALUE
SPECIFICATION->FIELD
SPECIFICATION->FIELD VALUE
SPECIFICATION->VALUE FIELD
RELATION-PATH->SPECIFICATION RELATION-PATH
RELATION-PATH->RELATION-PATH SPECIFICATION
RELATION-PATH->RELATION
RELATION-PATH->RELATION RELATION-PATH
QUERY->RELATION-PATH
QUERY->RELATION-PATH QUERY
```

This of course would have to be filled out in considerably more detail. The grammar still needs to deal with such problems

as determiners, quantifiers, question word like "what," negatives, verb phrases, prepositional phrases, and so forth, as well as clarifying the semantic criteria for the applicability of the above rules. The basic scheme, however, will remain the same because of the fact that we will always be mapping queries into a relational hierarchy.

3.1.3

In AQL, a word is put into a query language vocabulary by assigning a syntactic category and a semantic procedure. Basic syntax words like "the" will be defined within the AQL basic grammar, but vocabulary specific to a target data base will have to be defined separately. The syntactic categories for this are as follows:

RELN	A noun identifying a relation, e.g., "aircraft." This has syntactic feature MODF marking possible use as post modifier of a field expression.
VERB[RELN]	A relation expressed as a verb, with mandatory syntactic features, e.g., "fly."
ADJ	A relation expressed as an adjective, e.g., "big."
FLDN	A noun consituent of a field name, which may include more than one word, e.g., "name." This has three possible syntactic features: HEAD = can be head word of noun phrase. MODF = usually used as a modifier of head words. GENR = usually requires a modifier, implies HEAD.

FLDV	A field name expressed as a verb, e.g., "weigh."
FLDA	A field name expressed as an adjective, e.g., "wide."
TYPE	A general classifier in a field name that does not help to identify any relation, e.g., "distance."
TYPE[GENR]	A general classifier in a field name contributing no information at all by itself, e.g., "system." The syntactic flag is mandatory.
FKEY	A word used in many different contexts to distinguish different field names, e.g., "service." The syntactic feature HEAD marks possible occurrence as head word of noun phrase.
LST	A literal denoting a value for a field, e.g., "MIG-25." This has two possible syntactic features: HEAD = can be head word of noun phrase. MODF = usually used as a modifier of head words. Numbers need not be defined as literals; they are automatically recognized as category NUM.

A user will not have to be aware of the syntactic category of any word in a query. Defining words by syntactic category will be the job of the person who sets up a relational hierarchy and the translation tables associated with it. This person in turn, however, needs to know only the categories listed here, not all the various categories required for a query language grammar.

3.2 A parsing scheme

AQT input query processing is organized around a general

syntax-driven parsing algorithm for context-free languages [4], further augmented to accept syntactic and semantic restrictions on the applicability of rules of grammar in a given situation. This approach keeps the relative simplicity of a context-free scheme without being forced to recogonize only context-free languages. In fact, the possible restrictions on rules approach in power that of van Wijngaarden grammars.

The context-free part of the AQT parser is taken directly from the LINGOL system of Vaughan Pratt [9]. It is essentially a bottom-up (Cocke-Kasami-Younger) algorithm combined with a simulation of a topdown (Earley) algorithm to get the best features of both. It is driven by a grammar with syntax rules of the form X->Z or X->Y Z; the procedure is as follows:

Assume a sentence consisting of morphemes at positions 0 through L. Let [X,i,j] denote a phrase of type X comprising morphemes at positions i through j.

Begin at position 0 with the goal of S, the sentence start symbol for a grammar.

Let n be the current position. Generate phrases [Z,n,n] such that Z is a possible part of speech for the morpheme at position n and such that a phrase of type Z is compatible with a goal at position n.

Now get consequences of all newly generated phrases [Z,k,n] for current positionn n as follows:

- o For each rule X->Y Z

  If a phrase [Y,m,k-1] has already been generated and X is consistent with a goal at position m, generate a new phrase [X,m,n]

- o For each rule X->Z

  If X is inconsistent with a goal at position k, generate a new phrase [X,k,n]

- o For each rule X->Z W

  If X is consistent with a goal at position k, establish W as a goal at position (n+1).

Continue the above until all newly generated phrases ending at position n have been processed. Then advance to position (n+1) and repeat until the end of the sentence.

The consistency checks above are not necessary for correctness, but improve efficiency by assuring that no phrase is generated unless it would have been for a top-down as well as a bottom-up algorithm. A phrase of type X is consistent with goal W if an X-phrase can ever be a leftmost subconstituent of a W-phrase. The consistency relation can be established when a grammar is defined and stored as a Boolean matrix.

The parsing scheme defined so far is sufficient for any context-free language, but for natural language, it is rather

clumsy. It is inconvenient for handling grammatical relations like agreement and tends to require a proliferation of production rules for describing grammatical features in a language, like English. To make the scheme more workable for natural language, three restrictions can be added to the basic bottom-up parsing algorithm.

Syntactic Features These have already been described in Section 3.1.1. We can easily incorporate them into the AQT parsing algorithm by simply testing for the applicability of a rule for given subconstituents before proceeding. In a van Wijngaarden grammar, syntactic types and syntactic features would correspond to "proto-notions" and rules of grammar to "hyper rules". In effect, a rule of grammar is extended to become a family of related rules, allowing contextual linguistic dependence to expressed compactly.

Semantic Features

Semantic conditions for the applicability of a rule of grammar for natural language will tend to be less strict than syntactic conditions. We often want to grade the semantic acceptability of different rules in a given situation without necessarily having to reject any of them out of hand. To accomplish this, it is helpful for phrases to be associated with

sets of semantic features, similar to syntactic features but maintained in an altogether different way. We will allow special clauses to be attached to a rule of grammar; each clause will specify a condition on semantic features of phrases for the right side of th the rule and, if the condition holds, will also specify semantic features to be assigned to the new phrase generated for the left of the rule plus a semantic rating of the new phrase. This mechanism is helpful in choosing between alternate interpretations in cases of a single rule of grammar doing double duty.

Externally Defined Attributes

The LINGOL system uses LISP local variables to implement the manipulation of attributes in Knuth's scheme of semantics for context-free languages [7]. These local variables are defined in semantic procedures associated with individual phrases, which are executed at the conclusion of a parse to verify that it also semantically acceptable. With local variables having a scope of definition over a given phrase and all of its subconstituents, semantic procedures can communicate with each other regardless of the syntactic independence of their respective phrases. AQT takes a parallel approach with semantic procedures and local variables; it allows for variables to be declared, set, and tested within procedures and for a procedure to reject a parse if

specified conditions are not met. This arrangement provides AQT with most of its parsing power beyond that of the basic bottom-up scheme driven by context-free grammars.

All three of these restriction mechanisms are quite general, not at all limited to natural language applications. Our particular interest in natural language suggests that we might go even further in the development of the parser. This will make it less appropriate for certain classes of context-free languages, but will help it to be more efficient for languages like English, a prime consideration when working with minicomputers.

The main enhancement of the parser deals with right-recursive rules of grammar, important because natural language (except for the single case of Japanese) favor right recursion. The parsing algorithm as described so far would establish identical goals for each of the phrases of identical syntactic type in a right-recursive nesting, resulting in many extra phrases being parsed. We know, however, that only the outermost phrase in a right recursive nesting need be considered in a natural language. The parser therefore includes code to recognize this special case.

An enhancement is included also for one particular kind of left-recursive rule involving an inflectional suffix. The

stemmer automatically removes such a suffix as a distinct morpheme to be put back later by a left-recursive rule of grammar. The result of this with a left-to-right parsing algorithm as in AQT is that all the new phrases generated as consequence of the root of a word must be regenerated for the grammatical combination of the root plus a suffix. This special case is easily recognized, and by judicious relabeling of phrases, all regeneration can be eliminated.

These two special cases for recursive rules of grammar and the three mechanisms for restriction described above show the main things to note in order to get a broad idea of how the AQT parser works.

3.3 Text editor semantics for rewriting

Because the parser in AQT has to translate a natural language input query string into an intermediate query string, the semantic procedures for rules of grammar in AQT need a general ability to manipulate strings. As a result, semantic procedures in AQT are defined in terms of text editing primitives along with basic programming control structures. The primitives refer to two types of data objects: buffers in which strings are operated upon, and variables storing a single character, a

pointer to a string in a dictionary, or a pointer to a list of string pointers.

Every constituent of a query as derived in parsing is associated with a semantic procedure through the rule of grammar describing the constituent. Execution of procedures in top-down, with the procedure for the constituent corresponding to an entire query running first. This procedure will in general call the procedures for its immediate subconstituents and they in turn theirs, going on down to single words with procedures that can only insert strings into a buffer or set variables.

Prior to any execution, there is only a single empty buffer, with no variables defined. A semantic procedure can append to the current contents of a buffer or split off a new empty buffer and transfer processing to it temporarily; along the way, variables can be declared, set, and tested. Transferring processing to a new buffer masks all previous buffers, and declaring a variable masks previous variables of the same name. Buffers and variables are accessible to all procedures, but must be deallocated upon return from the procedure creating them, restoring access to previous buffers and variables. Upon conclusion of all execution of all execution, there is again a single buffer with no variables defined, but the buffer now will contain an output string.

Here is an example of some semantic procedures in AQT:
(lines preceded by ";" are comments)

```
rule NP->ELEMENT NP

     ;START NEW CLAUSE
     LINEFEED
     ;EXECUTE FOR "ELEMENT"
     LEFT-PROCEDURE-CALL
     ;EXECUTE FOR "NP"
     RIGHT-PROCEDURE-CALL
     RETURN

rule NP->ELEMENT
     ;START NEW CLAUSE
     LINEFEED
     ;EXECUTE FOR "ELEMENT"
     LEFT-PROCEDURE-CALL
     RETURN

rule ELEMENT->SPECIFICATION
     ;DECLARE VARIABLES
     VAR @Q
     VAR @R
     ;CHECK DEPENDENCE OF "ELEMENT"
     IF DEPEND=T
         GET @Q RELN
     ELSE
         CLEAR @Q
         END
     ;MOVE TO NEW BUFFER
     SPLIT-BUFFER
     ;EXECUTE FOR "SPECIFICATION" IN NEW BUFFER
     LEFT-PROCEDURE-CALL
     ;RETURN TO PREVIOUS BUFFER
     BACK
     ;GET RELATIVE PATH FOR SPECIFICATION
     ;IN RELATION HIERARCHY
     DIFFER @Q @R
     ;ADD CONTENTS OF NEW BUFFER
     MERGE
     ;SAVE CURRENT RELATION
     PUT @R RELN
     ;DEPENDENCE FROM NOW ON
     SET DEPEND=T
     RETURN

rule SPECIFICATION->LIT FLDN
     ;DECLARE VARIABLES
     VAR @Q
```

```
VAR *E
VAR *F
VAR *V
;EXECUTE FOR FLDN FIRST
RIGHT-PROCEDURE-CALL
;SAVE RESULTS
PUT @R TEMP
PUT *F FIELD
;EXECUTE FOR LIT
LEFT-PROCEDURE-CALL
;CHECK COMPATIBILITY OF RELATIONS
GET @Q TEMP
SAME @Q @R
;(FAILS IF NOT SAME)
APPEND [
;INSERT FIELD NAME
GET *E FIELD
JOIN *F *E
;(FAILS IF FIELDS NOT SAME)
APPEND=
;INSERT LITERAL
JOIN *V
;END SYNTAX FOR CLAUSE
APPEND ]
;SO THAT @R HAS ORIGINAL VALUE
GET @R TEMP
RETURN
```

dictionary entry "ROLE" FLDN

```
;RELATION AS PATH IN HIERARCHY
SUBSEQ @R AIRCRAFT
;FIELD NAME
POINT *F ROLE
RETURN
```

dictionary entry "INTERCEPTOR" LIT

```
;RELATION AS PATH IN HIERARCHY
SUBSEQ @R AIRCRAFT
;FIELD NAME
POINT *F ROLE
;VALUE IN FORM OF A STRING PATTERN TO MATCH
POINT *V  AN  I  /R
RETURN
```

The procedures here are simplified from those in an actual query facility, but illustrate essentially what is involved in

producing an intermediate query from an input query. The procedures associated with a basic query language grammar in general will work entirely with pointers relations, fields, or values, which will be set by procedures for dictionary entries. In this way, the same grammar can be used with different dictionaries; all that is necessary is for a dictionary to know what variables to pass information through. In the case of AQT, only three variables will be of concern: @R for a relational path, *F for a field, and *V for a literal value. The procedures for setting these variables will be simple enough to be generated automatically from input lists describing associations between literal values and fields and between fields and relations.

3.4 Recognizing dependence between parts of a query

The AQT intermediate translator incorporates special features for dealing with linguistic dependence in a query. The most simple kind that has to be considered is the implicit local dependence between two subconstituents coming together to form a single new constituent; in the intermediate form of a query, this has to be mapped into an explicit dependence path in a relational hierarchy that connects different fields. Several semantic primitives in AQT help to facilitate this: one allows a path from the top of a hierarchy down to a given relation to be

expressed as a list of pointers as relation name strings (SUBSEQ); one compares a list of string pointers with another and puts at the start of an intermediate query clause the path sequence of the first where it diverges from the second (DIFFER); and one compares two downward paths and fails if they differ, thus rejecting the current parse (SAME).

The dependent relational paths generated by these primitives will by design not be unambiguous in themselves because they will be relative paths requiring a context of interpretation. This ambiguity, however, turns out to be useful where a field name may be defined in several different relations; instead of listing all of the cases and trying to figure out which is the right one in the intermediate translator, we enter an ambiguous path and rely on intermediate query resolution in the target data base translator to choose the proper interpretation in the current context. This kind of postponement is convenient because the intermediate translator, having semantic procedures built around string manipulation, is poorly equipped to make a correct decision. The approach here allows the intermediate translator to remain fairly simple while not complicating intermediate query resolution all that much.

Another processing strategy along these lines is used in AQF for more global sorts of dependence in aspects of language like

pronoun reference, definite noun phrases, and some kinds of fragmentary queries. These kinds of language usage must be taken care of in the intermediate translator at least to the extent of reducing them into terms expressible within the AQT intermediate language. The limited semantic processing in the intermediate translator prevents doing a great deal, but it is possible to accomplish something nontrivial and thus simplify the problems of processing further down the line in the target data base translator. This is in fact an advantage of a multi-pass translation strategy.

To handle global linguistic dependence in AQT, a number of special global variables are defined. These are like the ordinary variables declared in semantic procedures in that they would hold same sorts of information, but they are defined outside any semantic procedure and in fact outside the intermediate translator itself since they must be saved between queries. The global variables keep track of the last relation and the last field referenced (RELN and FIELD respectively) as well as of the relational focus of a query as derived from syntax and of the kinds of references to counts. Global variables are accessible only through the primitives GET and PUT.

The usage of global variables is straightforward. In the case of a pronoun or an information request like "how many?"

where no relational information is given, this is simply filled out from the appropriate global variables and processing continues as before. There is no attempt to resolve global dependence in the sense of actually finding referents, though. This is left up to subsequent passes in the target data base translator.

3.5 Support facilities

The ACT front-end language processor has several built-in features that simplify the task of defining a query language for a user.

- Inflectional stemmer
  Words in a query language vocabulary may often appear with -s, -ed, and -ing inflectional suffixes. AQT includes a procedure that automatically removes such suffixes; it recognizes over 500 different cases of word endings, substantially more than similar procedures on most natural language systems. This means that only root forms of regular nouns and verbs have to be entered in a dictionary.
- Undefined literals
  Query words undefined in a dictionary can still be processed. This is helpful when a target data field

can take a large number of non-numeric values; AQT automatically takes care of undefined words corresponding to data field values when the field is explicit, and other cases can be handled with the addition of rules of grammar for the UNKNOWN syntactic category.

o String patterns

AQT has a built-in string pattern matcher that allows for concise intermediate language definitions of words corresponding to target data fields or values. This is capable of matching initial substrings, embedded optional substrings, and a combination of alternate string patterns. It is especially helpful for complex multiword field names or literal values that may appear in many different ways.

These kinds of features support a data base manager in setting up a query facility. This process is unseen by the data base user, but it is nevertheless important in determining the ultimate usefulness of the query facility. The ability for a data base manager to change or to add to a query language with a minimum of effort helps greatly in tailoring a facility to a particular set of users and in extending the facility when its applications grow. Although this is not as convenient as in some

systems where users themselves can mold a query language, it should work out much the same anyway, assuming that query language will eventually stabilize.

SECTION 4

Target Data Base Access

4.1 Data access sequences

An input query translated into intermediate form refers to a relational hierarchy serving as a logical model for a target data base. At this point, there are many ways to map the intermediate query references into an access sequence for the target data base; but for portability, it is convenient to adopt a table-driven scheme along the lines of the AQT parser for input queries. The necessary tables for this can be made external to the actual code for a query facility, keeping the code independent of any particular data base.

A correspondence between logical fields in a relational hierarchy and actual fields in records of a target data base is easy to establish in a table. There is, however, a problem in mapping over the kinds of dependencies defined between the different fields of a relational hierarchy; for example, the

connection between a key field and a non-key field at different levels along a path of relational hierarchy. Because the corresponding fields in the target data base may not be in the same record or even in two directly linked records, the generation of a data access sequence with the right dependence requires a little work.

The basis for access sequence generation comes from two observations about necessary correspondences between relational hierarchies and target data bases. First, if two fields are in the same relation, then the target data fields must be on the same record or on two records connected by a chain of one-to-one links. Second, if two fields are in hierarchically adjacent relations, then either the corresponding target data fields are either in the same record or one is connected to the other by a chain of one-to-one or one-to-many links. This all comes out of having set up relational hierarchies so that non-key fields are functionally dependent on key fields at the same level or above.

Relational fields that occur along the same downward path in a hierarchy are dependent, and so there should be a sequence of trivial, one-to-one, and one-to-many links connecting the target data records with the corresponding fields. To derive such a sequence, we will begin by interpreting the direct record links of a target data base in terms of a relational hierarchy. That

is, we will think of these links not only as connections between different record types, but also as implicit intra-relational and interrelational connections; a link will be treated as a way of moving from one coordinate of (relation, record type) to another coordinate.

This will all be part of a logical data model completely external to a target data base. We can change our interpretation without changing the target data base. In general, a given direct linkage may be used for moving between several different pairs of (relation, record type) coordinates; and of course, we can always choose to disregard certain direct linkages altogether with a given logical model. A classification of direct record linkages for a relational hierarchy will be stored in external intra-relational and intra-relational link tables read by a target data base translator.

To avoid combinatorial problems, inter-relational links will be defined so that they can always correspond to going down in a hierarchy; and a notion of sublevel will be established for intra-relational links to allow a similar downward restriction for them as well. This will assure that generated sequences will be forced in effect to move down paths in a relational hierarchy and eventually hit bottom. The relational hierarchy here models a user's perception of access to stored data.

4.2 A 4-pass access scheme

Interpretation of an intermediate query string begins with its conversion into a resolved internal form. This involves elimination of any ambiguity with relation names in clauses, the collection of field references for the same relations, and in case of an initial dependent clause, the merger of information from the intermediate query string with the resolved form of the previous query. This process has already been described in Section 2.

The procedure for generating a data access sequence from a resolved intermediate query will be to proceed backwards from fields in the query as follows:

- o Look up each relational field specified in the intermediate query and get the record type associated with it, the location and size of the data field in a record, and the type of data stored there. This information can be collected in an external field correspondence table keyed by field name and relation.
- o If the record type for a field is directly accessible in a target data base, such as through some hashing, indexing or

sequential access method, then we are done for that field.

- If a record type is not directly accessible, then we need to get to it via a direct link from another record type. To get this link we first look for any interrelational link or, if there is none, any intra-relational link for the given (relation, record type) coordinate and then continue as above for the new (relation, record type) coordinate. If a relational hierarchy is set up correctly, then there should always be a link when needed.

After applying this procedure for all fields in an intermediate query, the result is a branching sequence of accesses with a structure paralleling the dependence between fields expressed in the intermediate query. The structure will not be identical to the intermediate query, however, because a data relation may encompass several different data record types and different relations may be defined over the same record types. It is also convenient sometimes to introduce along an access sequence some additional logical relations not referenced in a relational hierarchy.

Each entry in the inter- and intra-relational link tables

will include an encoding of the actual access linkage to get from one record type to another; for example, hashing a secondary key or following a pointer or skipping some offset. There is a potential problem of encoding here in that there can be arbitrarily many types of linkages in general, but since only a finite number are used in any given target data base, our approach will be to have a few common linkage types predefined and allow a user to introduce more exotic types by linking in special code to handle them.

Retrieval of information with a data access sequence from an intermediate query will be through a special search procedure built into query facility. This will use the access sequence to traverse records of a target data base making comparisons of fields against given query conditions and extracting requested information where indicated. The result of this will be a series of index lists specifying record instances by type and relation that happen to match the conditions for an access sequence. Retention of relational identifiers as a distinction in index lists is necessary because record instances listed under different relational headings will be selected or rejected in general by different criteria and will call for different output.

Index lists need to be saved because natural language queries may contain pronouns or other linguistic constructions

that refer to results of a preceding query. The idea here is in effect to treat a dependent query of this sort simply as a continuation of the previous query, taking up in the traversal of a target data base where the other left off. It turns out that this requires little to be added to the basic data search scheme since dependence already must be handled within a single query anyway.

Output can be produced for a query as soon as a complete set of matching records instances have been collected for a data access sequence. In the case of multiple matching instances at a one-to-many link, our approach will be to collect all of them at the same time except for multiplicities at the head of data access sequences; this happens to be a convenient point to save partial results for a query when space for index lists is limited. As for the actual format of the output, this comes naturally out of the definition of relational hierarchies; each downward path in a hierarchy, and consequently the corresponding data access subsequence, can be printed out as a third-normal form relation.

Sections 5 and 6 will describe how the procedures outlined here have been implemented on th AQT demonstration system and will go into some key problems arising in them. On the whole, however, these procedures are fairly straightforward for all they

do and can be readily put into FORTRAN. The simplicity of the basic scheme is a distinct advantage, especially in comparison to what other systems have to do in order to implement similar capabilities.

SECTION 5

Special Problems

5.1 Generic secondary keys, null keys

In a target data base, certain values may be stored in several versions, with some kind of key distinguishing the various versions; for example, "high count", "low count", and "best count". The keys are typically generic in that they are not necessarily unique to a given class of values; and there may or may not be an actual target data base allocation for storing the actual key. Proper resolution of these generic keys require knowledge of how a target data base is actually set up.

It is advantageous to avoid having to resolve generic keys while parsing. This keeps input query processing independent of any target data base application and also simplifies the parsing procedure. During query translation in AQT, generic keys are kept around as bracketed components of their associated field names until the generation of data access sequences for the data

base. The keys are then looked up in a special external table keyed by relation to determine how they actually refer to the target data base.

Closely related to generic keys are qualifiers so general in a given text that they carry no information at all; for example, "Russian" in a data base of Soviet aircraft. These null qualifiers can be handled in a data base dictionary by assigning them to a special syntactic category that is skipped over in parsing. Note that a logical model in selecting a given subset of a target data base for access may result in changing an actual key into null qualifier.

5.2 Numerical computation

When getting numerical information from a target data base, a statistical summary of values is often more convenient than a straight list of them. Accordingly, comprehensive data base access facilities usually build in standard statistical functions like mean, standard deviation, maximum, and minimum. In AQT, such functions are computed by a special module that does all the work of identifying a function by name, of collecting the values for calculation, and of displaying the results. The module can be tailored for a particular application without affecting the

rest of the code for query translation.

The implementation of a statistical module is straightforward except for a possible complication with naming. It is possible for field of a target data base to contain a statistical function as part of its name; for example, "maximum weight" could be an actual stored value. In this case, the target data base translator needs to recognize that no computation is necessary, although a user need not know one way or the other.

The procedure in AQT for handling function name components of a field name is similar to the one for handling generic keys. To avoid cluttering up the parser with information about a target data base, the resolution of the function name is postponed until the generation of data access sequences when field names have to be looked up. If a function name is not found as part of a field name, the corresponding function will be computed.

A general arithmetic capability is part of the long range plan for AQT. This could be implemented by special functions specified in the intermediate query language that emulated the operation of a desk calculator. Other computational capabilities such as for generating graphical displays, data smoothing, and deriving robust descriptive statistics in the manner of Tukey

[11] are also possible.

5.3 Arrays

A field in a relational hierarchy may actually correspond to an array of values in a target data base. This complicates AQT data access because it involves another level of data extraction when individual array elements as well as an array itself have to be manipulated. The problem here is to define a general access scheme that is independent of any target data base.

The AQT approach to arrays will in effect be to translate the multiplicity of array elements in a single record into a multiplicity of virtual records each with a single scalar array element value. This is accomplished by including an array element offset value in the index lists of record instances matching conditions of a given query. The array offset serves to identify the particular array element associated with a given virtual record instance.

Although this scheme requires linking in special user-supplied procedures to compute array elements offset values, overall it is fairly straightforward. The only serious restriction is that an array field value in a relational

hierarchy must be the only field of some relation so that an array offset is unambiguously associated with that field; but this is natural since arrays typically are named, giving us a way to put it into a relational hierarchy. Having virtual record instances costs only extra space in index lists; no extra I/O is needed.

5.4 Ellipsis

An elliptical expression in natural language is an abbreviated kind of linguistic usage that requires expansion within some context before it can be understood. For example,

What is the length of the Flogger?

The Foxbat?

where the second query is elliptical and must be understood in context as "What is the length of the Foxbat?" This kind of usage makes a language highly efficient, and so natural query language facilities almost always make some attempt to support it through incorporation of special procedures.

In the AQF approach, however, no special procedures are

required. Ellipsis is handled entirely through existing mechanisms for resolving reference. An elliptical query is treated exactly like a query containing an anaphoric expression; in their intermediate forms, an elliptical query is merged with the resolved preceding query to produce a new resolved query. This greatly simplifies an AQT query facility.

5.5 'OR' condition

Disjunctive "OR" conditions in queries are implemented two different ways in AQT. The simpler way is to allow more than one value to be given in a field specification of a query, letting a pattern matcher try each value in succession against a target data field during a data base search. This would require only the addition of a few rules in a query language grammar and a straightforward extension of the pattern matching procedure in AQT.

More general disjunctive conditions in queries, however, must be expressed over sets of clauses, requiring elaboration of the control structures in target data base translation. The approach in AQT here is to take advantage of procedures already used for handling coreference. A series of clauses expressing different "OR" conditions are treated in the same way as series

of dependent clauses except that a flag is set to allow a restricted search on failure to change into an unrestricted search.

Each set of clauses for single disjunctive condition is processed separately in target data base translation, but with display of results put off until all clauses have been processed. This is only slightly more complicated than the other procedure for disjunction, requiring only a few changes at the top level of query processing and the addition of a branch in the code for searching a target data base.

5.6 Purging a context

In processing a query dependent on a preceding query, we need to start from the record instances matched by the preceding query, but we have to be careful about which instances we actually retain. To see why this is so, we might consider the following simple relational hierarchy:

```
AIRCRAFT [nato name]
    :
    :
    : . . CONFIGURATION [type]
```

where AIRCRAFT to CONFIGURATION is one-to-many, and the intermediate queries

```
AIRCRAFT(y/n?)[nato name=Flogger]
. CONFIGURATION [type=tactical support]
(.)

. AIRCRAFT(y/n?)CONFIGURATION [type=air superiority]
(.)
```

In the first query, we get a record instance for CONFIGURATION [type], but we do not want to restrict our search of CONFIGURATION [type] to this instance in the second query, for then no match will ever be possible.

The rule here is that record instances corresponding to a relation in a hierarchy should be purged if there are query markers along the hierarchical path for the relation and if these are all above it. This is a selective purging distinct from that involved in resetting a context upon getting an independent query. It has to be done after results are displayed, but before the resolved intermediate form of the preceding query is lost. The decision to reset a context in general can come only after the preceding intermediate query has been merged with the new query.

5.7 Conversational postulates governing responses

Although we can always give a data base user exactly what was asked for by interpreting queries literally, this may not actually be what the user wanted. In ordinary speech among people, literalness is never strictly enforced because we can usually tell when a literal response is inappropriate because we know what is typically expected in a given situation; for example, appropriate responses to the question "May I ask your name, please?" include "No" and "John Smith", but not a simple "Yes". These kinds of expectations can get extremely complex for ordinary speech, but in the narrow context of interactive data base access, it is fairly easy to build some of them into query interpretation to make it more hospitable to a user.

In AQ1, we can in fact do some things without having an elaborate model of expectations for a data base user.

o In yes/no or how-many queries, a user will frequently want a full retrieval of information rather a simple straight answer; for example, in

  "Do any fighters carry the ATOLL missiles?"

  it is likely that the designations of the fighters are also wanted. We can in general tell for sure when a full retrieval is called for, but we can easily enough prompt a user after making a straight answer. Although this makes a user do a bit more, it saves having to

re-enter a query if the full retrieval was wanted.

- o When the number of matching record instances in a target data base fails to equal a count specified in a query, it pays to go through a retrieval anyway so as not to waste work already done. For example, in the query

  "Are there two aircraft carrying the ACRID?"

  we would want to respond even if there was only one aircraft.
- o On full retrievals with many matched records, we will ask the user whether a complete listing is really desirable. A user typically will not want voluminous output at a terminal.

Procedures for ellipsis can also be mentioned here.

The rule in AQT here is always to give a user the benefit of the doubt in a situation. None of the features above involve any theoretical difficulties, and all are easy to implement. They are important, however, in that they make a query facility easier to work with but yet they are an aspect of natural language often neglected in so-called natural language systems.

5.8 On-line data base documentation

With natural language, it is possible for a user to enter an intelligible query with no references to fields either as search conditions or as requested information; for example, "Tell me about aircraft.", where "aircraft" is a relation name in a hierarchy. This query could be processed like other types of query, but the results would probably not be what the user wanted: the target data base interpreter would simply go through the entire data base and return all the mandatory key fields above the point in a relational hierarchy marked by the input query. This is a drastic kind of information request even for fairly small data bases, and on the whole, it seems reasonable to allow a user to make such a request only in an explicit way.

On the other hand, we do not want to disregard a query without fields entirely or to print on diagnostic message because the user probably entered the query in good faith, given that data base structure is assumed to be transparent. A reasonable interpretation of a general query without field references is that it is actually a request for general information about a data base itself rather than for a complete enumeration of the data base. In this way, queries of this type turn out to be a convenient way to implement on-line documentation for a data base or at least for a user's logical model of the data base.

Because a query without field references marks a relation,

AD-A079 626 PATTERN ANALYSIS AND RECOGNITION CORP ROME N Y F/G 5/2

ADVANCED QUERY TECHNIQUES.(U)

OCT 79 C P MAH, J M MORRIS F30602-77-C-0161

UNCLASSIFIED RADC-TR-79-260 NL

2 OF 2

ADA 079626

END

DATE FILMED

2-80

DDC

it seems appropriate to respond with information about that particular relation name. For example, in the case of "aircraft", we might print the names of its key fields, names of relations immediately below in the hierarchy and the percentage of a data base pertaining to aircraft. A message of this sort can be stored in an external file for each relation name of a hierarchy and retrieved when needed.

SECTION 6

A Demonstration System

6.1 Purposes

The AQT demonstration system was implemented to show that an effective natural language query facility could be built around the simple notion of a relational hierarchy. The basic AQT algorithms themselves were straightforward enough so that we could have kept to presenting a case for them on paper, but the immediate reality of the demonstration system is much more compelling, especially for the potential users of a natural language query facility. Although the demonstration system represents only a fragment of a full query facility, it shows what can be accomplished even on a relatively small machine like the DEC PDP-11/70.

The demonstration system also served as a major development tool for the design of a query facility. We have built several different versions with extensive instrumentation for experiments

with various aspects of query translation. Almost all work was done in FORTRAN with a FLECS preprocessor so as to approximate the conditions under which a query facility could eventually be implemented. This gave us an idea of the amount of computation involved in query translation and the amount of space required for it. A DEC PDP-11/70 proved to be adequate for an implementation, though careful organization of the demonstration system was required for it to run within a 16-bit address space.

We have derived most of the basic data and control structures of a natural language query facility through evolution of the demonstration system. Here, we shall describe the demonstration system itself as currently implemented and move on from this to a discussion of how eventually to build a full prototype query facility.

6.2 Basic structures

There are two categories of data structures defined in AQT: internal list structures produced as intermediate results during the various stages of query translation, and external tables supplied by a data base administrator to define a logical model and its correspondence to a target data base. A description of these categories of data structures fairly well characterizes how

AQT works, since the algorithms operating on these structures are relatively straightforward.

6.2.1

The principal internal data structures of AQT are the (1) the parse tree for an input query, (2) the resolved intermediate query expressed also as a tree, (3) the data access sequence for target data records, (4) the index lists of record instances matching conditions of a query, and (5) the lists of target data fields explicitly or implicitly requested for output in a query.

The parse tree for an input query is a standard type of phrase-structure description consisting basically of binary subtrees. The subtrees are built up of phrase nodes of the following form:

1 index of the rule of grammar generating a phrase
2 syntactic category of phrase
3 starting position of phrase is query
4 pointer to left descendant phrase node list
5 point to right descendant list, if any
6 semantic plausibility rating of phrase
7 encoded usage count
8 syntactic features

9 semantic features

10, 11 miscellaneous pointer links

In the demonstration system, each of the above node elements is a single byte long except for the rule index, which is two bytes long. A descendant node list will contain a single node except in case of ambiguity, when it will more generally contain multiple phrase nodes of the same syntactic category and syntactic and semantic features, generated from different rules of grammar. This scheme localizes ambiguity as long as possible in a phrase-structure description of an input query.

The parse tree defines an execution sequence for semantic procedures associated with rules of grammar. The result of this is an intermediate query expressed as a string. The intermediate query string is first printed at a user terminal as a check; then it is passed to the target data base translator, where it is converted into a resolved form with all relation names replaced by explicit references to a relational hierarchy and all references to fields of relation collected together.

A resolved intermediate query is a tree of linked nodes having the following format:

1 relation id number

2 ancestor link

3 descendant link

4 sibling link

5 count specification

6 encoded query markings

7 pointer to field list

8 type of link from ancestor

An element of a field list has the format:

1 pointer to field name

2 length of field name

3 pointer to value specification

4 length of value specification

5 link to next element

In a FORTRAN implementation, each item above would be an array of integers. Both links and pointers would be array indices; null links and pointers would be a zero value. Field names and value specifications will be character strings in a byte array.

The resolved intermediate form of the preceding query is used as a context of interpretation when the current qury is dependent involving anaphoric reference or ellipsis. The basic procedure for resolving an intermediate query is to process a clause at a time, appending the results to that of preceding

clauses, if any; the generalization for anaphoric reference or ellipsis is essentially to take the previous query in resolved form as the starting point for processing the first clause of a dependent query.

From a resolved intermediate query, we next generate a target data record access sequence, describing the order in which we will read in record instances from a target data base and make comparisons against data fields.

Data access sequences are generated as tree structures with nodes as follows:

1 data base record type
2 relation id number
3 ancestor link
4 type of record linkage
5 link field offset in preceding record
6 link field length
7 units for record data field
8 type of record data field
9 data field offset in record
10 data field length
11 count specification
12 encoded query markings
13 pointer to predicate for data field

14 pointer to list of matching record instances

15 descendant link

16 sibling link

In FORTRAN, this would be an integer array. Links and pointers would be array indices, with null value of 0.

AQT uses a data access sequence for the retrieval of record instances from a target data base. Those record instances matching conditions specified in the access sequence are saved on index lists. The head of an index list has the following format:

1 target data base record type

2 relation identifier

3 count of matching record instances

4 pointer to first instance

Record instances are chained together in a list structure having nodes with the following format:

1 record number for instance

2 pointer to predecessor instance along data access sequence

3 pointer to next instance on index list

4 back pointer to index list head

5 array element offset

The key point to note here is that separate index lists for each different relation is kept for any record type. Both index list heads and index list nodes are defined as integer arrays. The demonstration system keeps all index lists in main memory; a prototype query facility would them out piecewise onto secondary storage for each iteration through a data access sequence.

Once we have a full set of matching target data record instances for a data access sequence, we need to extract the information to be returned in response to a query. This is described through a required output list associated with index lists according to record type and relational identifier; the list is derived from the field name references of the original input query and from a table of mandatory fields for output enumerated by record type plus relational identifier. Output list nodes will have the following format:

1 byte offset of requested field in target data record
2 byte length of field
3 pointer to field name, if any, in original query
4 length of field name
5 data type of field
6 pointer to next required output list element
7 pointer to function, if any, to be computed on field

This is defined as an integer array.

Output is composed starting with a record instance corresponding to an endpoint in a data access sequence. We then follow back the chain of predecessor record instances, extracting required fields along the way. Results are printed as a row of a multi-column table, going from right to left in order of extraction. Since following predecessor links corresponds to going upward in relational hierarchy, this ends up with more general fields appearing at the left.

6.2.2

To drive the query translation process, we will have various tables describing the syntax and vocabulary of an input query language and the structure of the target data base pertinent to a user. These tables are implemented as external files on the AQT demonstration system for maximum flexibility, but for speed, they could be linked directly with the code of a query facility. There are eight principal ones altogether in AQT.

(1) grammar table

Syntactic rules of the form X->Y Z are represented as a byte array.

1 syntactic category X

2 features to be set on generating phrase with rule

3 syntactic category Z

4 usage count

5 positive preconditions on Y features

6 and negative preconditions

7 positive preconditions on Z features

8 and negative preconditions

and syntactic rules of the form X => W also as a byte array

1 syntactic category X

2 features to be set on generating phrase with rule

3 positive preconditions on W features

4 and negative preconditions Rules X=>Y Z are stored on

lists according to syntactic category Y. Rules X->W are stored on lists according to syntactic category W. Preconditions apply to the features of phrases being combined through a rule of grammar to generate a new phrase. Section 2.3 on parsing describes these matters in more detail.

(2) dictionary table

vocabulary specific to a target data base will be kept in an external dictionary. An entry will have the form:

1 word string

2 syntactic category of word

3 syntactic features to be set for word as a phrase

4 special semantic flags

5 special semantic flags

6 path in relational hierarchy expressed as string

7 field name string, if any

8 literal valuee string, if any A word string will have a fixed length of 24 characters. Path, field and value strings will be variable length with a final delimiter, having combined length of 132 characters. All other fields will be a single byte long.

(3) relational model

This table describes the structure of a relational hierarchy, the relation names associated with them, and the default record type for each relation. Table entries have the format:

1 relation name string

2 default record type

3 index of next relation up in hierarchy

4 multiplicity of hierarchical linkage A relation name is a fixed length string of 18 characters. The remaining fields are integers.

(4) field correspondence table

This maps a field name for a given relation into an actual data field of a target data base record. Each table entry has the format:

1 field name string

2 relation index

3 target data record type

4 type of data in field

5 offset of field in target data entry

6 length of field

A field name is a fixed length string of 40 characters. All other fields are integer values.

(5) generic secondary keys

When a generic key corresponds to an actual target data value, it is listed in this table under a given relation index. If not listed, it must be part of a field name. The format of a table entry is:

1 generic key string

2 value string

3 relation index

4 target data record type

5 offset of field in target record

6 length of field

The key and value strings have a fixed length of 12 characters. The value string is substituted for the key upon a match. All other components of an entry are integers.

(f) relational link tables

The inter-relational and intra-relational link tables will have entries of the same format:

1 present record type

2 present relation index

3 next record type going backwards

4 next relation index/sublevel

5 encoding of linkage type

6 offset of linkage area in nextt record type

7 size of linkage area A potential problem here is the encoding of linkage types, since there can be arbitrarily many of these in general. Our approach will be to have certain common linkage types predefined in a query facility and to allow a user to define more exotic types by linking in special code to handle them. All other components of an entry are integers. Sublevels are defined only for intra-relational links, taking the place of the next relation index.

(7) record access

A record access table describes how to gain access to target data records of a given type. Table entries have the following format:

1 record access method

2 size of record in bytes

3 file name string

In the demonstration only standard FORTRAN I/O is assumed, restricting possible access methods to sequential or direct encoded as a single ascii character. A file name is a 22 character fixed length string. The record size is an integer value.

(8) mandatory fields

This table lists record key information that should always be printed to aid a user in interpreting output values. Table entries are integer arrays

1 target data record type

2 relation index

3 key field offset in target record

4 key field length Mandatory fields apply only to record type and relation index combinations occurring before explicitly requested information in a data access sequence.

6.3 System configuration

The AQL demonstration system in its RSX-11M implementation is organized into five passes, each being a separate overlay. The processing of any query proceeds sequentially through all the passes. The first pass contains the AQL intermediate translator comprising a syntax-driven query parser along with a query language grammar; this reads a query in English form and rewrites it in intermediate form still as a string. The intermediate form is printed for inspection, and if approved, it is passed onto the largest data base translator comprising the remaining four passes. These four passes successively carry out the work of:

1 converting an intermediate query string into a resolved intermediate query with dependence on preceding queries made explicit,

2 generating a target data base access sequence from the resolved intermediate query and from various translation tables

3 traversing the data base access sequence and retrieving record instances that match conditions along the access sequence

4 deriving a format for output and printing out fields requested from matched record instances.

Each of the five passes is essentially the execution of a single principal algorithm. The one for the first pass of parsing and intermediate translation has already been described in detail in Section 3. Here, we shall go into the algorithms for the four passes of the target data base translator. Unlike the very general algorithm of the first pass, which can be applied to a wide variety of problems, these are specific to AQT and in fact constitute the heart of AQT. Most of the AQT effort was devoted to their development.

6.3.1

The algorithm for resolving intermediate queries is quite simple, consisting mainly of an inner loop and an outer loop. The overall structure is as follows:

Step 1 If a query is dependent, start from the resolved intermediate form of the preceding query; otherwise, discard the preceding results and start anew.

Step 2 Get successive clauses of the current query until an end mark is reached.

Step 2.1 Extract successive tokens from a clause until exhausted.

Step 2.1.1 For the case of the token being

1 a relation name, look it up in the relation name table and try to find a place for it in the resolved intermediate query where processing left off last. If no place is found and we are at the start of a dependent clause, go up a level in the resolved intermediate query and try again. When a place is found, allocate a new node if there is none already.

2 a query mark, encode it and store it at the current place in the resolved intermediate query

3 a field specification, split it up into its field name and its logical condition and store a link to these at the current place in the resolved intermediate query.

4 exit with an error if the above fails

Step 3 Apply the criteria of Section 2.2 to determine whether to retain target data record instances from the preceding query.

6.3.2

The algorithm for access sequence generation consists of an inner and outer loop with an iterative subprocedure:

Step 1 Scan the nodes of a resolved intermediate query in order of allocation (i.e. straight interation with no attempt at a tree traversal).

Step 1.1 If there are field references for a node, then do for each field.

Step 1.1.1 Look up the field name in the field correspondence table to get the record type associated with the name and the current relation.

Step 1.1.2 Allocate a new acces sequence node for the field.

Step 1.1.3 Look up the current relation and record type first in the inter-relation link table, or if this fails, in the intra-relation link table. Get the predecessor record type and relation along an access sequence.

Step 1.1.4 If there is already an access sequence node with the same record type and relation as the new node, append the new node after it with a trivial link. Otherwise, if there is already a node of the predecessor record type and relation, append the new node after it with the link type from the link table.

Otherwise, if the new node can start an access sequence, add it to the list of starting nodes. Otherwise, allocate a predecessor node, link the current node to it, and repeat this step with the new predecessor node.

Step 1.13 If there are no field references for a relation with a query mark, look up the default record type for it, allocate a node will a null field, and proceed from 1.13 as with an actual field.

Step 2 Optimize the data access sequence so that nodes of the same record type and relation as a predecessor node are listed first as those coming after it.

The access sequence generated by this procedure is in the form of a forest of trees.

6.3.3

The search and retrieval pass of AQT has the most complex of all the algorithms in AQT. The procedure can be broken into two pieces: a part for proceeding down an access sequence, and a part for backing up along an access sequence on hitting an endpoint or exhausting all possibilities for satisfying a

condition of an access sequence.

The "down" part is as follows:

Step 1 Read the next target data record of the type specified in the current node of an access sequence. Go up on failure.

Step 2 If their is a field condition, compare it against the data record. Go up on failure.

Step 3 On a match, allocate a record instance node if doing an unrestricted search, not starting from instances collected previously.

Step 4 Go up upon reaching the end of an access sequence. Otherwise, save the current record instance and go down, starting over at Step 1.

The "up" part is as follows:

Step 1 If there is a match for the access sequence returned from,

Step 1.1 Increment the count of matches for the current point in the sequence

Step 1.2 If the search is unrestricted, link in the record instance for the current place in the access sequence.

Step 1.3 If the record instance is one of a multiple

set, get the next one and go down.

Step 1a If there is no match,

Step 1a.1 Purge the record instance at the current point in the access sequence

Step 1a.2 If the record instance is one of a multiple set, get the next one and go down.

Step 1a.3 Otherwise, set the match flag if any record instances at all are still matched at the current point in the access sequence.

Step 2 Back up in the access sequence

Step 3 If coming from a match and there are access sequences starting from the current point, go down the next one.

Step 4 If at the beginning of the access sequence, the procedure is finished. Otherwise go back to Step 1.

This procedure starts at the beginning of an access sequence, going down. A search is restricted if there is already an index list for a given record type and relation; note that index lists can be saved from a preceding query in the case of co-reference.

6.3.4

The formatting and printing algorithm is as follows:

Step 1	If this is a query without specified fields, print out data base documentation and quit. Otherwise, scan the data access sequence to
Step 1.1	Get all endpoints of an access sequence.
Step 1.2	Get all query markers and fields for which output is requested.
Step 1.3	Verify that there are matching record instances for requested fields.
Step 2	If there are no requested fields and it is not a yes/no query, quit with an error message.
Step 3	If it is a yes/no or a how-many query, respond according to matching record instances and prompt the user for full output if there were matches.
Step 3a	Otherwise, format and print full output as described in Section 4.2 if there were matches.
Step 4	Delete record instances from index lists if they correspond only to points of an access sequence after a query mark.

6.4 Setting up a data base

To test ADF, we put together a Soviet fighter aircraft data

base along the lines of a possible implementation for an SEF intelligence application. The choice of subject was influenced by an earlier effort to bring up a similar data base on the REL system, but our basic approach was different. We deliberately avoided trying to construct the data base to fit AQT; the design was done independently by a co-worker unfamiliar with the workings of AQT.

The test data base currently contains information about 29 Soviet fighters collected from open sources into four basic record types. For the most part, the information consists of aircraft attributes such as wing span, fuselage length, and empty weight along with keys such as service name and NATO name; these were implemented as simple fields of a single record type keyed by aircraft and presented no major problems in query translation. Other types of data had much more complex structure, however, and presented a major challenge.

Maximum speed versus altitude information had been generated hypothetically for aircraft by fitting a simple single-maxima approximating curve to the known maximum speed, the altitude at which maximum speed was attained, and the service ceiling. By taking points at altitudes at sea level and up by increments of 10,000 feet, we obtained a speed profile of the aircraft as an array of mach numbers. The array was fixed with a size of 11

numbers, but many aircraft would have fewer actual values. In the test data base, the array for an aircraft was stored in a single record type linked by a pointer from the main aircraft record type.

The armament configurations for an aircraft were lists of the quantities of weapons that might be carried on a mission. The number of configurations could be different for each aircraft, and so could the number of types of weapons for each configuration. Because weapons tended to be common to several configurations of an aircraft, the lists of weapons for an aircraft were merged intot a tree structure to save space; a downward path in a tree would represent a single onfiguration. Tree nodes were another record type, and weapons data were collected into a fourth record type to avoid duplicating information in the tree; pointers to these records were stored in tree nodes along with the associated counts.

Having gotten a target data base, the next step was to construct a relational hierarchy to model it logically. The hierarchy finally settled upon was:

```
AIRCRAFT
  :
  :
  : . . ARMAMENT
  :        :
  :        :
  :        : . . CONFIGURATION
  :                 :
  :                 :
  :                 : . . WEAPON
  :
  :
  : . . CREW
  :
  :
  : . . DIMENSION
  :
  :
  : . . ENGINE
  :        :
  :        :
  :        : . . IDENTIFICATION
  :
  : . . FUSELAGE
  :        :
  :        :
  :        : . . DIMENSION
  :
  : . . IDENTIFICATION
  :
  : . . PERFORMANCE
  :
  :
  : . . WEIGHT
  :
  :
  : . . WING
           :
           :
           : . . DIMENSION
```

The relations here were chosen because they are useful in categorizing the data in the Soviet aircraft data base. We also found it convenient to define some hidden relations not appearing in a hierarchy but occurring in access sequences to establish

alternate paths having different mandatory fields for output; this is is used in printing armament configurations (see Appendix B). These hidden relations, however, are not actually part of a logical model.

With a relational hierarchy as a logical model, we could identify data items of interest in the Soviet aircraft data base by assigning field names to them and attaching them to appropriate relations in the logical model. The model itself and the correspondence between the model and the target data base were established by setting up the various necessary translation tables (see Appendix B). These tables were written out as files to be read back during query translation.

Speed versus altitude profiles and armament configurations could not be handled entirely through the translation tables. It was necessary to insert code for them in an AQT special linkage module, which is called to handle unusual kinds of data and data linkages. A procedure for speed versus altitude was supplied to determine how many valid values there were and to initialize AQT array element sequencing accordingly. A procedure for armament configurations was supplied to unravel configurations from their tree representation and to write out the lists of weapons into a temporary space for sequential searching. With these procedures, the special linkage module would allow access to the first and

subsequent record instances associated with speed data or armament without having to know how this was accomplished.

For the display of information, it was also necessary to define certain virtual fields that do not correspond to any target data. We needed some way to identify individual configurations, marking where one weapons list ends and another begins, and some way to show the altitude values implicitly associated with speeds. This was done by defining virtual fields for data records, indicated by a negative offset; instead of being extracted from a target data record, these fields would be computed by a call to a special module containing code for that purpose.

In the AQT demonstration system, the special linkage and virtual field modules contain all the code dependent on the Soviet aircraft test data base. The remainder, comprising over ninety percent of the program lines for query translation and including all the major AQT algorithms, is independent. The idea of a portable AQT facility readily adaptable to different data bases is therefore reasonable; and in fact the demonstration system shows us how this can be accomplished with a table-driven scheme.

6.5 Performance

Although the AUT demonstration system was not designed to be a full query facility, it does allow a user to type in a broad range of natural language queries and to get correct answers back from an actual data base. The system is quite easy to use, especially in comparison to the data access facilities usually found on medium-scale processors. (See Appendix A for an example of an actual query session.) The capabilities of the demonstration system at present are limited by two factors: the size of the grammar driving the intermediate translator and unimplemented features in the target data base translator. The intermediate translator now runs with only about 300 grammatical rules, while basic logical and arithmetic operations line negation, quantification, and general comparison of stored numerical values are unavailable in the demonstration target data base translator.

The full demonstration system in its full FLECS FORTRAN version currently has a total task image size of about 120K bytes (K = 1024). This is overlaid extensively so as to run in a maximum address space of 64K bytes on the DEC PDP-11/70. The system now fills up all the available address space while running, but we should be able to reduce this requirement in the prototype query facility by taking advantage of an external

dictionary, reducing the amount of instrumentation, and foregoing some of the array checking and tracing options during FORTRAN compilation.

The demonstration has a response time of under 10 seconds for queries directed at the Soviet aircraft data base. Most of this is in waiting for data base records to be read in during the search and retrieval phase; the target files and translation tables were all stored on an RK05 cartridge, which has a fairly slow access time. Parsing input queries takes relatively little time, almost always under a second; this turns out to be much less important than delays introduced from the use of a remote terminal with a slow communications rate. The time required to read in translation tables from disk files is also relatively small, mainly because the tables themselves are small.

The main cost factors in running the demonstration system are its size and its target data base I/O. On a small machine, the system can take up a significant fraction of main memory and consequently load down operations, although the impact of this should be no more than that of running a large FORTRAN compiler. The data base I/O problem will be much more serious, requiring careful optimization of programs to avoid unnecessary access to secondary storage. This is, however, as much a problem of the target data base as of a query facility; if the data base is not

designed to make the required kinds of access efficient, then the query facility should not be faulted much. Here it may ironically be undesirable to make access to a data base easier for users.

The demonstration system as written in FORTRAN is about as portable as can be expected in using mostly standard FORTRAN I/O and avoiding calls to an operating system except for overlaying. The main problem in portability will be in differences between dialects of FORTRAN. The demonstration system employs OPEN and CLOSE statements and declarations specifying the sizes of data items such as LOGICAL*1; these were necessary to implement the kinds of data manipulation required in AQT, but will not exist in many FORTRAN compilers. The situation is improved considerably by the FORTRAN 77 standards, but for full portability, a system would probably have to isolate much of its data manipulation in interchangeable modules. The issue of how much portability to aim for will have to be addressed in the subsequent development of a full query facility.

SECTION 7

Comparison with an S&T Data Base Communication Facility

7.1 The User Communication System

This section will review the functional design of the FTD User Communication System (Usercom), a component of the FTD update program. The purpose of this review is to identify features of the Usercom design that differ in approach from those of AQT, and to indicate possible advantages or disadvantages of those features.

Since Usercom exists only as a functional design, there have been no operational tests to determine the effectiveness of the approach. For this reason, the following review must be taken as suggestive, rather than definitive. It is intended primarily to highlight those differences in design that might aid in making AQT more responsive to the needs of potential users, and it is not intended as a critique of the Usercom design.

7.2 Evaluation Criteria

At the request of RADC, a number of criteria were suggested for the evaluation of user interface languages for data base management systems. In somewhat modified form, the criteria suggested were these:

7.2.1 Ease of Learning

A primary advantage of a user interface language should be the ease with which it can be acquired and remembered by the novice or casual user. The time required to learn the operation of a sequence of function keys should be comparable to the time required to learn a natural or artificial language for accessing the same data to a comparable level of effectiveness.

Note that some period of training will be required for any interface language, even though implementors of natural language systems sometimes assume that users require no training, since the language is "natural" to them. Such an assumption could actually make it more difficult for the user to employ such a system, if it includes undocumented requirements, restrictions, and idiosyncracies absent in human languages. A well-designed selection of user-oriented function keys could then be easier to

use than a "natural" language.

7.2.2 Power

A system is better to the degree that it can process more of the queries that can ordinarily be expected in a specific application. Conversely, if the user finds it impossible to extract information clearly contained in, or implied by, a data base, then the query language will be less useful in that application.

The last words should be emphasized, since additional power is undesirable if it will be unused. For example, a complex parser will be superfluous if nearly all the queries take a very small number of syntactic forms. In such cases, the use of a relatively small number of function keys may be considerably easier than the use of a sophisticated system for analyzing natural language. A large vocabulary is not useful if most of the words are never actually encountered. Unneeded power will simply be a waste of resources.

7.2.3 Ease of Use

An "ideal" natural language interface should accept a

problem statement in any words that the user requires. A formalized artificial language may require extensive reworking of the problem statement in order to fit the specified formats. Although this criterion is somewhat more "subjective" than the first two, it is important; users will become discouraged with a system that demands extensive work in getting the problem formulated correctly.

7.2.4 Correctness

This evaluation criterion is intended to locate instances in which the user enters a query with a straightforward English language meaning, but which is given some other meaning by the system. To take a rather artificial example, suppose that the user asks: "How many MIG-17s do the Vietnamese have? What is their range?" If the system were to interpret "their" as referring to the Vietnamese rather than to the MIG-17s, it could produce output that was wildly incorrect. Such errors may be more serious in a natural language system than in an artificial language system, since the chance for misinterpretation of natural language may be greater.

In an important article, Christine Montgomery has pointed to one of the major weaknesses of the natural-language approach:

its notorious inability to deal with the fractured grammar, misspellings, and general ill-formedness of genuinely natural language, the language that actually appears in inputs to real systems. (Montgomery, C>A>, "Is Natural Language an Unnatural Query Language?", Operating Systems, Inc., Woodland Hills CA 91364). This point of view clearly provides some of the incentive to develop Usercom as a system organized around designated function keys, rather than around natural-language input.

In terms of the evaluation of a functional design, this criterion would apply to the likelihood of user error, given that the software functioned as specified. What is the probability that the user would type misspelled words, incorrect syntax, wrong function keys, and other erroneous input?

7.2.5 Naturalness

The language should permit the user to employ a natural form of the language. The REL system, for example, sometimes requires rather stilted English: "Who are the shippers of shipments whose cargo type is general merchandise?" (Thompson, Bozena Henisz, and Thompson, Frederick B., "Rapidly Extendable Natural Language," Proceedings, ACM National Meetings, 1978, pp. 173-182.) Thus a

natural language system could be less "natural" for the user than a set of clearly-defined function keys, if the latter bore some correspondence to the user's own definition of the problem.

7.2.6 Helpfulness of error messages

When the system fails to interpret a query, the response should be one that assists the user in reformulating it (cf. Codd, E.F., "Seven Steps to Rendezvous with the Casual User," IBM Research Report RJ 1333, Jan. 17, 1974). An uninformative message, such as "Eh?" is better than no message at all; but the system should be designed so that failure to produce an acceptable interpretation should retain sufficient information to permit a reasonable response to the user. Because it is sometimes difficult for a system to locate the precise source of error when an interpretation fails, it will be important to evaluate the correctness of error messages in an implemented system. In the evaluation of a functional design, of course, correctness would be difficult or impossible to determine; nevertheless, a review of proposed error responses will indicate the likelihood that they will be of value to the user.

7.2.7 System capacity

Since many question-answering systems have been developed on an experimental basis, with very small data bases, it is important to determine whether they can be applied to data bases of significant size. (Note, for example, that Winograd's well-known SHRDLU operated with a vocabulary of only 200 words (cf. Winograd, Terry, Understanding Natural Language, New York: Academic Press, 1972).)

In the evaluation of a functional design, there should therefore be some indication that the design can be applied to a data base of significant size -- enough to warrant the cost of implementation.

7.2.8 Visibility of operation

Does the system permit the user to inspect the query at intermediate points -- say, between the time the query has been translated into an internal language and the time that actual searching begins? For the more experienced user, it will be important to be able to locate errors in retrievals before they occur; in particular, to locate errors before they soak up hours of wasted machine time.

At the same time, for the casual user with a relatively, simple query, the opposite virtue is needed: transparency of operation. For the casual user, the inner working of the system is of little interest. It is only when something has gone badly wrong that such a user will want to step through the details of the retrieval, under the guidance of an experienced user or system manager, to locate the precise point where an instruction was misinterpreted.

7.2.9 Self-documentation

It should be possible to review hardcopy output from a retrieval session, after several months, and to be able to interpret the goal of each query. It would be unfortunate to receive several beautifully formatted, carefully plotted charts or graphs without any information concerning their purport. A major advantage of a natural language system should be the ease of reading directly, from the queries and the responses, exactly what the user was trying to find out. Similarly, a system like Usercom should provide output that is easily interpreted by the user.

7.2.10 Abbreviations and Shortcuts

For many users, typing may be a difficult and lengthy process. Is it possible to abbreviate the query in such a way as to permit much shorter input? For example, can minor function words ("a", "the", "is", etc.) be omitted from the query? Can the user or the system manager introduce abbreviations for frequently used terms and phrases? Will the system provide pre-defined function keys, for example?

7.2.11 Notational Idiosyncracies

Does the language require a large number of notational devices and special formats or symbols, which might detract from concentration on the problem? (Cf. Sammet, Jean E., Programming Languages: History and Fundamentals, Englewood Cliffs: Prentice-Hall, 1969).

7.2.12 Resource Requirements

For a given data base, what are the requirements in terms of primary and secondary storage, central processor time, terminal time, and peripheral time? Can the system be implemented on small or medium-sized equipment, or does it require a very large

computer mainframe? Does it require a dedicated processor, or can it be time-shared with other processes?

7.2.13 Portability

Is the system available in a standard language available on a variety of machines likely to be found in the intended installations? Is the coding highly machine-dependent, or does it make use only of widely available features? Is special-purpose hardware required? Does it normally operate with widely-used terminal equipment?

The following items are regarded as somewhat beyond the state of the art for operational systems using large data bases. They are, however, in use for smaller, experimental systems and should be considered typical of many desirable options.

7.2.14 Spelling corrector

An effective editor is definitely not beyond the state of the art, and should be regarded as a positive element in any implementation. In addition, techniques for modifying minor misspellings (and informing the user of changes proposed) have been used in correcting student programs, and may be applicable

to data base management systems. When a spelling corrector fails to locate a word in its vocabulary it would not only print an error message, but would also suggest some possible alternative spellings. If they were acceptable to the user, they would be substituted automatically.

7.2.15 Cooperation with User

"Cooperative" is used to describe a system that attempts to anticipate the user's information needs. (Cf. Kaplan, S. Jerrold, "On the Difference Between Natural Language and High Level Query Languages," Proceedings, ACM National Meetings, 1978, pp. 27-38). Consider the following answers to the question: Q. Which Cambodian bases that can service heavy bombers can also service tactical fighters? A1. None A2. No Cambodian bases can service heavy bombers.

The second reply obviously provides the user with an important piece of information that is ignored in the first reply. (Cf. Belnap, Nuel D., Jr. and Steel, Thomas B., Jr., The Logic of Questions and Answers, New Haven: Yale, 1976; and Lehnert, Wendy, The Process of Question Answering: A Computer Simulation of Cognition, New York: Halsted, 1978). Determining the presuppositions of the user's query would be an important

feature of a system making a significant contribution to the ease of making queries.

7.2.16 Approximations and Near-Misses

(Cf. Siklossy, Laurent, "Impertinent Question-Answering Systems: Justification and Theory," Proceedings, ACM National Meetings, 1978, pp. 39-44). Consider the following pair of responses:

Q. What Chinese troops are now active in Cambodia?

A1. None.

A2. None, but the Third Division is stationed five miles from the border.

The second reply helps to give the user information that might well be relevant, but which did not appear in the query as it was first formulated.

The last three features maybe taken as examples of work that is actively being pursued in research in artificial intelligence, but which is not yet ready for incorporation in large data base

management systems. They seem to require resources that far overbalance their apparent benefits; and there is some likelihood that they will provide intolerable levels of noise--i.e., unwanted information. These and other experimental features should nevertheless be taken into consideration in evaluating competing data base access languages.

7.3 Review of Usercom

The sixteen criteria that have been suggested here may be used as the basis for a review of the functional design of Usercom, with emphasis on useful features for incorporation into the Advanced Query Facility. The paragraphs in the following review have been numbered to correspond to the evaluation criteria.

7.3.1

Usercom may be somewhat easier to learn than might appear at first, to the extent that it follows the normal sequence of activities now used by analysts in producing FTD's reports. For new analysts, the training procedures could well be built upon the use of automated aids in retrieving, manipulating, and formatting required information; such a training program would

not be intrinsically more difficult than a program that relied wholly on manual procedures.

In the absence of user experience with Usercom, it is particularly difficult to determine the ease of learning. It nevertheless seems to present several problems from the point of view of human factors:

- o Usercom requires the analyst to use a sequence of special purpose function keys, alternating these with names that are entered through a typewriter-like keyboard. The sequence of entries is rigid, resembling the required sequences of a programming language. Because of this rigidity, users adverse to learning a programming language may also find it unpleasant to comply with the sequence of actions required by Usercom. They may not like a rigid, somewhat arbitrary form of input.
- o Although Usercom reduces the amount of typing required of the user, it does not eliminate the need for typing. On the contrary, the user must first press one or more function keys, then must transfer to the typewriter keyboard, then move back again to the function keys, and so on throughout the input session. For an experienced typist, this is likely to be rather frustrating, because he cannot keep his

hands in the normal position required for touch typing, and must relocate his hands before each typewriter input. For such a typist, locating and using a function key may be more difficult than typing a brief function name at the keyboard.

- The use of prompting messages on the screen will be helpful to the beginning user in preparing the input in the correct sequence. It may be difficult, however, for the user to look at the screen, then look downward and refocus on the keyboard to locate the correct function key, then look up at the screen again, and so on. Differences in lighting between the screen and the keyboard may cause some eyestrain over a long period of use. An experienced typist would prefer to leave his hands in the normal position, without looking at them, rather than have to shift his vision from screen to keyboard between each use.

Like other human factors considerations, these comments would have to be tested in an actual implementation of Usercom to determine their applicability. From the point of view of AQT, however, it does not appear that there are features of Usercom that could be adapted to improve ease of learning. At worst -- pending tests with actual users -- the AQT natural-language approach does not appear to be significantly more difficult to

learn.

7.3.2

The power of the Usercom system may be limited by the number of function keys available in the particular hardware configuration chosen. However, the most serious apparent problem is in the update of the system as new functions are required and old functions are replaced or eliminated. Since new function keys must be designated, new templates will be required. Usercom, however, plays a strictly limited role: it is not a general purpose user interface, but an interface tailored specifically for access to STIS by FTD analysts. It requires precisely as much power as is needed to extract and combine data from the STIS data base and to prepare reports that needed by FTD. More power than this would be wasted. In addition, after some user experience with the system, it could be expanded to include functions that were needed but not included in the initial breadboard system.

The Usercom goal is therefore rather different from that of AQT, in that AQT is intended to provide an easily portable system for use by persons with a variety of backgrounds and needs in accessing data bases of differing designs. Usercom would be

optimized for access to the STIS data base, for a user group limited to FTD analysts. Since goals of the two projects are quite different, comparisons in terms of power would be inappropriate.

7.3.3

The ease of use of the Usercom approach should be a primary advantage. Without operational testing, of course, it would be premature to attempt to determine user acceptability. Nevertheless, there is a strong argument to be made in favor of the use of function keys, rather than the requirement that users type function names on a typewriter style keyboard. This approach shoud appeal particularly to those users who find typing difficult.

Arguments for the Usercom approach suggest a need for facilities within AQT that will help to improve it ease of use. Primarily, such facilities are intended to reduce the amount of typing and thereby to make it easier for the non-typist to use. Specifically, AQT might consider:

- o Use of abbreviations. It should be possible to use just one or two letters of a command word, rather than the whole word. For example, "f" might

abbreviated "find", and "hm" might abbreviate "how many".

- o Tolerant syntax. AQF should do no more parsing than is necessary to set up the required search routines. Omission of a function word ("the," "and," "is) should not cause program failure. The user should thus be able to employ a very concise format for queries.
- o Since misspellings may occur, it will be important to be able to deal with them without causing the user inconvenience. For this purpose, a good text editor will be of assistance. In a more ambitious approach, a spelling corrector might be employed, to check inputs against a list of common misspellings, or to locate other words with spellings that are similar to the input. Because of the likelihood of error in a spelling corrector that deals with unrestricted natural language, it will be essential to secure the user's agreement before making changes. For example:

  Is Chine still using MIG-17 fighters?
  "Chine" not found. Do you mean "China"?
  yes

  At this point, an editing routine would substitute the suggested spelling into the query and proceed.

  Use of aids such as these could make the AQF natural-language approach as easy to use as the Usercom approach.

7.3.4

By "correctness" is meant the ability of the system to interpret a user's intentions in ways that are satisfactory to the user, ways that carry out the user's intentions. The criterion reflects our dissatisfaction with operation of one major natural-language system in which the system sometimes failed to understand queries and produced incorrect parsings. In such cases an incorrect answer would be returned to the user without warning.

The Usercom approach represents one way of avoiding this kind of error. The user is given the responsibility of preparing a query in a rigid, unambiguous form, which the system can interpret and execute in only one way. When errors occur, they are more likely to be user errors, caused by a failure on the part of the user to employ the proper operators in the correct sequence.

Errors in a natural-language system are likely to be of another order. In such a system, the user uses whatever phrases and forms he would normally use in formulating a query. The system then has the responsibility for translating these into a formula meaningful to the system. This translation process is difficult, and is subject to the errors that normally occur in

natural language -- the result of ambiguities in words (equivocation) and sentence structure (amphiboly). Because of the complexity of natural language, there will always be sentences meaningful to the user that are incorrectly interpreted by the program. The problem of pronoun reference, for example, is particularly difficult.

In comparing the AQT approach with that of Usercom, then, we are comparing alternatives with very different sources of potential errors. In Usercom, errors are most likely to occur as the user attempts to translate an information requirement into the language of the system; in AQT, errors may occur when the system interprets an input incorrectly. Performance in Usercom is likely to improve as the user learns more about system requirements and is better able to tailor queries to fit the system. In AQT, performance improves with time as plausible errors are identified and the system is improved. Experience with both systems, Usercom and AQT, is required to determine the degree to which errors can be reduced or eliminated under either approach.

7.3.5

The Usercom approach to system design defines "naturalness"

in terms of the natural sequence of activities that the analyst performs. It is intended to provide options in an ordering and a format that correspond to the way in of analysis has dictated the design of the system.

"Naturalness" in AQ1 has a somewhat different meaning. Here, it refers to how a human user normally phrases a query. The use of relational hierarchies to model a data base was was an attempt to provide a close approximation to the way humans normally formulate their queries. Because of these differences in approach, the degree to which the naturalness of one system can be transferred to the other is not great.

7.3.6

The helpfulness of error messages would be an important criterion for review of a running system. At this time, Usercom exists only in the form of a generalized system description, while AQ1 has been implemented in a demonstration system. In neither case is there sufficient experience to be able to determine the helpfulness of error messages.

7.3.7

Again, there are no firm data upon which to base estimates of system capacity. Both Usercom and AQT have been designed for eventual application in the production environment, where data bases are potentially very large. It will be assumed that both systems will be capable of dealing with very large data bases in their operational versions.

7.3.8

Visibility of operation should be interpreted as meaning the ease with which problems are located -- particularly problems in the interpretation and execution of queries. This problem does not appear to arise in the Usercom approach, since the primary locus for errors would be the user's interpretation of a query in the language of the system. In AQT, the approach has been to maximize the transparency of the system to the user, who need not be aware of the structure of the data base, or of the specific procedures that are used to extract information from it.

Both Usercom and AQT, then, rely on transparency rather than visibility as a desideratum. Further experience with both systems should show whether there is actually a need for greater

visibility, i.e. the degree to which the user can see exactly what the system is doing.

7.3.9

Usercom does not appear to be self-documenting in the sense of this criterion. It would be rather hard for an inexperienced user to determine the intent of a six-month-old record of interactions with the system -- if, indeed, the user dialogue would be retained at all by the system.

AQT is designed in such a way as to be self-documenting, since the full dialogue will provide even the casual reader with a clear picture of the original query and of the system's response. This documentation will be of value when it is necessary to determine the meaning of lengthy tables, charts, or other output that could not easily be regenerated.

7.3.10

Abbreviations and shortcuts were suggested in item ?.?.? above as methods of reducing the need for lengthy typing. Usercom represents an approach where all major system functions are supplied through function keys, representing an extreme form

of abbreviation. AQT has a very tolerant syntax, permitting the use of abbreviated, telegraphic inputs as long as they suffice to disambiguate the query; no rigid adherence to English grammar is required. (This approach provides a response to the position taken in Christine Montgomery's paper, "Is Natural Language an Unnatural Query Language?")

7.3.11

AQT is designed to be free of notational idiosyncrasies that might detract from concentration on the problem. The natural-language approach is intended to free the user from the need to employ a specialized syntax or set of symbols.

Usercom uses a rather different approach, in which the specialized syntax consists of a sequence of keys to perform elementary functions of the system. A programming option permits the user to define functions or macros that combine these primitive operations.

This criterion was suggested as a response to comments by Jean E. Sammet, in her Programming Languages: History and Fundamentals, where she points to the use of curious or difficult conventions within the languages under review. Neither the

Usercom nor the AQT approach seems to be subject to the criticisms that she raises. Nevertheless, idiosyncrasies might well appear in operational implementations of either system; the lack of experience with the systems again makes application of the criterion difficult.

7.3.12

In terms of resource requirements, both approaches seem quite similar, since the major resource requirements for both of them would be the storage of formatted data and the search mechanisms required to access the data. The front end for both systems, given the modest requirments of the Usercom approach, would show the largest difference. AQT takes the responsibility for translating the user's query into a set of commands for search, retrieval, transformation, and output. Usercom places responsibility upon the user for the first of these functions, the translation of an information requirement into the restricted syntax of the system. The greater power of the AQT approach will require proportionately greater system resources.

7.3.13

The portability of the AQT system will be one of its primary

advantages. It is designed for rapid implementation in a variety of hardware environments, for accessing a wide range of formatted data bases. Usercom is intended for a single type of data base.

Another characteristic of the Usercom approach that will limit its portability is the use of LISP as the system language, where AQT has used FORTRAN. Although both languages are about the same age -- dating from the late 1950's -- the number of LISP implementations is considerably smaller than the number of FORTRAN implementations. In addition, there are fewer programmers trained in LISP than in FORTRAN. As a result, Usercom is likely to be much less portable to new machines than is AQT. (None of these comments are intended to suggest that FORTRAN is superior to LISP as a language for implementation of systems like Usercom; on the contrary, LISP is a well-designed, flexible language particularly appropriate for such applications. PASCAL would be another language superior to FORTRAN to be considered).

The remaining evaluation criteria were suggested as typical of extensions of the natural language approach for the development of user interfaces. None of them should be regarded as within the state of the art for large data base management systems.

7.4 Summary

Several elements of the Usercom approach have been reviewed and emphasized here:

- The close attention to the actual operations of FTD analysts, to their information needs, and to the need for producing formatted output have been emphasized in Usercom and are important considerations in the development of AQT.
- Comments concerning the difficulties that some users will find in using a typewriter keyboard should be reviewed to insure that AQT is easy to use.
- The need for an effective implementation language, such as LISP, should be considered. A specialized language like PASCAL might also be reviewed.
- Comments concerning the problems that users find in producing syntactically-correct inputs, correct spellings, and other non-technical details, should be reviewed carefully. In particular, it should be possible to make corrections in an input without retyping the entire line.
- More generally, a close attention to human factors will be a major consideration for AQT development.

SECTION 8

Conclusions

8.1 Status and results

Our experience in building a demonstration system for AQT indicates that a portable natural language data base query facility is not only feasible, but also well within the domain of present established data base and natural language processing techniques. The most significant aspect of the demonstration system is its overall simplicity in contrast to its capabilities. The simplicity is reflected in the fact that such a system could be implemented in FORTRAN on a DEC PDP-11/45 and that we have been able to describe all the important aspects of the system here in this report. An eventual prototype query facility of course would have to go considerably beyond the demonstration system, but its basic framework should still be the same.

Most of the code written for the demonstration can be readily adapted to a FORTRAN implementation of a prototype query

facility. As noted already, the choice of FORTRAN here is solely on the basis of its near-universality although its portability may well be questioned; in practice, we found it troublesome to work with even with a FLECS preprocessor to aid in maintaining a top-down structured programming discipline. If a specific application called for implementation in a modern programming language like PASCAL, it would be fairly straightforward to translate from existing FORTRAN code into it; and in fact the resulting code should be much improved since the demonstration system has had to employ convoluted means in FORTRAN to come up with the equivalent of pointers for strings, structured data types, recursion, and local variables.

If need be, the present demonstration system itself could be applied to provide natural language access to an existing target data base. The system was designed to avoid the appearance of being tied to a specific test data base; we can redirect it to other target data bases by simply changing its external translation tables, assuming that the structure of the target data base is fairly orthodox. The demonstration system would of course constitute only a part of a prototype query facility, but it would nevertheless allow for experimental use of natural language access to actual data bases.

Experimental use of AQT will be essential to further

development. We need to get more experience with the kinds of data bases where natural language access would be appropriate and with the kinds of patterns of access that might be expected. We also need to identify specific information problems of both users and their organizations where new technologies might be directed. Because the success of an information system must in the end be judged by how well it is accepted by its users, it is crucial that they be able to have some say in how that system should turn out.

8.2 Evaluation criteria

The AQT approach is not concerned with developing the ultimate natural language system. Such a goal is of theoretical interest, but it is only tangential to the practical problem of facilitating data base access. Although a user can be impressed by the sophistication of a natural language system, it will be of little consequence if it requires riding roughshod over budgets in order to get one. Because of this economic reality, the emphasis in AQT has been on simplicity, with natural language left to fit in where possible.

This limitation on natural language power, though, turns out to be only minor in the context of the data base access problem.

The kinds of responses that we can make to a data base query are actually few in number, making it hardly worthwhile to be especially subtle in query processing. The resolution of reference is AQT, for example, is accomplished through extremely simple means, particularly in contrast with the lengths that some "intelligent" natural language systems go to; these simple means, however, seem to be entirely workable.

An AQT facility will work best with data bases that have highly regular structures and that deal with one particular subject area. It is designed for selectively displaying the contents of a data base without any automatic support of inference; it is possible to retrieve information not explicitly stored in a data base, but this must be done by including special procedures that define virtual data fields in effect. An AQT facility thus will be most useful in applications where the problem is the accessibility of online data and not its interpretation. This probably a reasonable strategy given that human beings are typically better at interpretation than machines while finding difficulty with the mechanics of data access.

The effective use of an AQT facility will require a certain amount of training. First, a user must know in a general way what is contained in a target data base because language processing in AQT depends a great deal on queries being

domain-restricted. Second, the user must also understand that an AQT facility is basically not intelligent; it can only respond directly to queries and even at this must ultimately fail as queries grow increasingly complex. The user, however, need not be familiar with the actual structure of the target data base or with the actual operations involved in query translation. The data base manager will require the most training.

An AQT facility is designed for easy installation and maintenance. Changes to a target data base can be accommodated in most cases by updating translation tables. Extending a query language vocabulary is a simple process; extending a grammar is more difficult in that it requires some expertise in linguistics, but this can also be done as a table update. The identification and correction of actual errors in code is aided by the organization of the query translation process into distinct passes and by the avoidance of complex algorithms.

The eventual goal of AQT is to produce a natural language system that is flexible but yet reliable enough to run effectively with applications in the real world. A persistent difficulty with natural language is that it still lacks credibility; for despite its supposed advantages, it is still hard to convince persons responsible for assembling practical information systems that true natural language access is a

serious alternative. We want to show with AQT that natural language is no longer a hothouse curiosity, but a basic tool to solve real problems.

8.3 Areas for further work

Because the AQT demonstration system was built primarily to show possibilities we were concerned not so much with developing specific aspects of the system as with estalishing the overall concept of table-driven query translation. There remains a considerable amount of work to do for the implementation of a full prototype query facility; the problems involve both enhancements of basic capabilities already in the demonstration system and altogether new capabilities necessary to support true natural language access.

The principal enhancements that seem called for are:

o numerical computations on fields

The demonstration system now computes sums, averages, maxima, minima of fields in a rudimentary way. This could be elaborated further to incorporate more statistical functions and to allow arithmetic operations on fields and the results of functions. One

interesting possibility is to build in tools for exploratory data analysis in the manner of Tukey [11], including the various graphical displays that are an intrinsic part of such analysis. There are many things we could do here; the major problem here will be that of deciding what is most appropriate for a general query facility.

- reference

  The current procedures for reference do not take into account the actual form of a query; they note only the intersection of the field specifications for the current and preceding queries. The scheme actually works out quite well, but it has the limitation of not correctly handling certain kinds of terse query sequences where consecutive queries have no explicit field specifications in common. It seems more reason-able to extend the reference procedure here rather than simply to forbid such sequences.

- grammar

  The current AQT grammar consists of a set of rules accumulated to test the demonstration system. There are deficiencies in handling conjunction and some rules, such those for sequences of field names, could be formulated better. The basic grammar will have to be reworked and extended for use in a prototype query

facility.

o indexing matched record instances

The demonstration system scheme for retrieving record instances and displaying results is adequate for a small test data base, but in general, it will require too much main memory for index lists. In the prototype query facility, retrieval and display of results will have to be interleaved, allow index lists to be read in and written out incrementally.

Some new areas to be explored include:

o negation

Negation in general can be exceedingly complex to handle in natural language, but because it is descriptively necessary at times, a query facility should accept it at least in a limited form. This will require major changes along the entire process of query translation.

o subspecies and variants

It is sometimes helpful for a logical data base model to allow subgrouping of data by certain keys; for example, in computing a mean, we may want to add in a single value for a key subgroup instead of taking them individually. The APF string pattern matcher used on ASCII keys addresses this problem somewhat, but a more general

approach is needed.

o sorting and grouping of results

In the demonstration system, results are displayed strictly in order of retrieval; but it is generally helpful for a user to have output sorted or grouped by various keys. This could be accomplished in a fairly straightforward way by adding another pass to the query translation process.

o interfaces with multiple data bases

Because a user sees data only through a logical model, it should not matter how many different data bases that data is drawn from. Two possibilities arise here: we can generalize the AQT record access procedure to allow a single logical model to refer to several data bases, or we could allow a user to switch from one logical model to another.

8.4 Plans

Current plans for AQT call for the development of a prototype query facility in FORTRAN. It will be implemented first on a DEC PDP-11/70 under the RSX-11M operating system and then probably taken to a VAX-11/780 under VMS running in 32-bit native mode. The move should allow the portability of the system to be assessed. Work in this area is scheduled for completion in

October of 1980.

As a test of the capabilities of the prototype query facility, we will try to get a large data base of current interest for an experimental application. In setting the system up, we plan to go through all the formal procedures that an actual data base manager would go through: identifying the target data to be accessed, constructing the tables for query translation, and actually generating a load image of the system from distribution sources.

To the extent that it is possible, we will get actual information users to try the system out, either by bringing it up on a processor available to the users or by arranging for communication from remote terminals over telephone lines. This would allow us to see how natural the system really is and also to let users help in guiding its course of development. It is hoped that the relative ease of setting up an AQT facility will make potential users more receptive to participation in such testing.

8.5 Summary

The philosophy behind AQT was that the natural language

aspect of the problem was actually less important than the data base aspect. Being able to express queries in the form of English is of little help if a user cannot figure out what to ask in a given situation. So instead of thinking about grammars and dictionaries first, we started by considering how to make data bases as transparent as possible so as to make access to them easier.

The major difficulty presented by data bases is that its structure is typically determined by purely technical considerations such as minimizing the amount of storage used, optimizing certain access paths, and exploiting storage devices. Because none of this is pertinent to a person who simply wants to retrieve information, the trend has been to make this invisible through imposing increasingly abstract logical models on top of data bases. The notion of relational hierarchies comes from this process of abstraction taken to an extreme: the naming of data items as well as their structure is made invisible.

The use of a relational hierarchy as a logical data model suggests a simple and practical way of interpreting natural language queries. Such queries are first mapped as an intermediate step into references to a relational hierarchies; these references can then be mapped into references to a larger data base through translation tables describing the

correspondence between an intermediate model and a data base. This approach does not require a large machine for an implementation; in fact, it can be programmed in FORTRAN on a DEC PDP-11/70.

The relative simplicity of the AQT puts it within reach of many information users for whom natural language access may otherwise be unavailable. Because the code for AQT is in FORTRAN it should be runnable on most machines large enough too support a data base system. The AQT facility is self-contained; it requires no additional software to go with, and it can as in the case of the demonstration system work directly on ordinary files without a separate data base management package.

An AQT facility is designed to go out to the user. Setting it up is easy, and it can readily evolve as a user gains experience with it. Most of the work of tailoring a facility to a target data base will be in deriving a relational hierarchy as a logical model and in constructing the translation tables to go with the model. Experimentation with the facility will involve very little risk on the part of the user.

The development of AQT is an attempt to make natural language a realistic solution to data base access problems. Natural language is important not only in that it can expedite

certain things now being done but also in that it opens up possibilities for bringing the ultimate users of information closer to the sources of information. Natural language can serve as a common access method to many different data bases at the same time, making their aggregate contents immediately available for analysis or decision making. The ADT effort is only a small step in this direction, but it is a realistic one and should help pave the way for other systems yet to come.

APPENDIX A

A. A Session with the Demonstration System

```
RUN AQT

*** AQT DEMONSTRATION SYSTEM ***

QRY>TELL ME ABOUT AIRCRAFT.

AIRCRAFT(?)
(.)

@1 AIRCRAFT
THIS IS A SOVIET FIGHTER AIRCRAFT DATA BASE,
DERIVED FROM UNCLASSIFIED PUBLICATIONS.  IT
DESCRIBES CREWS, FUSELAGES, WINGS, ENGINES,
ARMAMENTS, AND PERFORMANCE.

QRY>NAMES OF AIRCRAFT?

AIRCRAFT
. AIRCRAFT(?)[NAME=*]
(.)

NAME
-------------------------
FIREBAR        YAK-28P
FIDDLER        TU-28P
FLAGON-E       SU-15
FLAGON-D       SU-15
FLAGON-C       SU-15
FLAGON-B       SU-15
FLAGON-A       SU-15
FLAGON         SU-15
FISHPOT-C      SU-9
FISHPOT        SU-9
FOXBAT-B       MIG-25
```

4. A Session with the Demonstration System

```
FOXBAT-A       MIG-25
FOXBAT         MIG-25
FLOGGER-C      MIG-23U
FLOGGER-B      MIG-23B
FLOGGER        MIG-23
FISHBED-J      MIG-21PFMA
FISHBED-F      MIG-21PFM
FISHBED-D      MIG-21PF
FISHBED-L      MIG-21MF
FISHBED-K      MIG-21MF
FISHBED-J      MIG-21MF
FISHBED-C      MIG-21F
FISHBED        MIG-21
FARMER-C       MIG-19SF
FARMER-D       MIG-19PM
FARMER-D       MIG-19PF
FARMER         MIG-19
FRESCO         MIG-17

QRY>WHAT IS THE NATO NAME OF THE MIG-19 AND WHEN DID IT ENTER
SERVICE?

AIRCRAFT(1!)[SERVICE NAME=MIG-19]
. AIRCRAFT(?)[<<NATO>> NAME=*]
(.)

<<NATO>> NAME
--------------------------
FARMER          MIG-19

. AIRCRAFT
. AIRCRAFT(?)[WHEN IT ENTERED <<SERVICE>>=*]
(.)

WHEN IT ENTERED <<SERVICE>>
-------------------------------------------------------
1955                           MIG-19        FARMER


QRY>WHAT IS THE AVERAGE LENGTH OF AIRCRAFT WITH A WING SPAN
GREATER THAN 25?

AIRCRAFT
. DIMENSION(?)[(AVERAGE) LENGTH=*]
. WING DIMENSION[SPAN>25]
(.)

                               LENGTH
-----------------------------------
```

```
YAK-28P       FIREBAR       71.04
TU-28P        FIDDLER       85.00
SU-15         FLAGON-E      68.00
SU-15         FLAGON-D      68.00
SU-15         FLAGON-C      68.00
SU-15         FLAGON-B      68.00
SU-15         FLAGON-A      68.00
SU-15         FLAGON        68.00
SU-9          FISHPOT-C     55.00
SU-9          FISHPOT       55.00
MIG-25        FOXBAT-B      69.00
MIG-25        FOXBAT-A      69.00
MIG-25        FOXBAT        69.00
MIG-23D       FLOGGER-C     55.13
MIG-23B       FLOGGER-B     55.13
MIG-23        FLOGGER       55.13
MIG-19SF      FARMER-C      48.88
MIG-19PM      FARMER-D      48.88
MIG-19PF      FARMER-D      48.88
MIG-19        FARMER        48.88
MIG-17        FRESCO        36.33

AVE LENGTH= 60.87


QRY>CONFIGURATIONS OF THE MIG-21MF?

AIRCRAFT[SERVICE NAME=MIG-21MF]
. ARMAMENT CONFIGURATION(?)
(.)

-----------------------------------------------------------------
MIG-21MF      FISHBED-L     *CFGN*      2 UN BOMB       UKN
                                        2 UN BOMB       UKN
                                        4 AS ROCKET     UKN
                                        1 UN CANNON     UKN
                            *CFGN*      2 AA MISSILE    ATOLL
                                        1 UN CANNON     UKN
                            *CFGN*      2 AS ROCKET     UKN
                                        2 AA MISSILE    ATOLL
                                        1 UN CANNON     UKN
                            *CFGN*      4 AA MISSILE    ATOLL
                                        1 UN CANNON     UKN
MIG-21MF      FISHBED-K     *CFGN*      2 UN BOMB       UKN
                                        2 UN BOMB       UKN
                                        4 AS ROCKET     UKN
                                        1 UN CANNON     UKN
                            *CFGN*      2 AA MISSILE    ATOLL
                                        1 UN CANNON     UKN
                            *CFGN*      2 AS ROCKET     UKN
```

```
                                         2 AA MISSILE  ATOLL
                                         1 UN CANNON   UKN
                              *CFGN*     4 AA MISSILE  ATOLL
                                         1 UN CANNON   UKN
MIG-21MF     FISHBED-J        *CFGN*     2 UN BOMB     UKN
                                         2 UN BOMB     UKN
                                         4 AS ROCKET   UKN
                                         1 UN CANNON   UKN
                              *CFGN*     2 AA MISSILE  ATOLL
                                         1 UN CANNON   UKN
                              *CFGN*     2 AS ROCKET   UKN
                                         2 AA MISSILE  ATOLL
                                         1 UN CANNON   UKN
                              *CFGN*     4 AA MISSILE  ATOLL
                                         1 UN CANNON   UKN

QRY>WHAT IS THE COMBAT RADIUS OF SOVIET AIRCRAFT CARRYING ATOLL?

AIRCRAFT ARMAMENT CONFIGURATION WEAPON[NATO NAME=ATOLL]
. PERFORMANCE(?)(1!)[<<COMBAT>> RADIUS=*]
(.)

                              <<COMBAT>> RADIUS
-----------------------------------------------
MIG-21PFMA   FISHBED-J          1183
MIG-21PFM    FISHBED-F          1183
MIG-21MF     FISHBED-L          1183
MIG-21MF     FISHBED-K          1183
MIG-21MF     FISHBED-J          1183
MIG-21F      FISHBED-C          1183

QRY>DO ANY AIRCRAFT CARRY THE ACRID MISSILE?

AIRCRAFT(Y/N?)
. ARMAMENT CONFIGURATION WEAPON(1!)[CLASS=MISSILE,NATO
NAME=ACRID]
(.)

YES.
FULL OUTPUT?
NO

QRY>HOW MANY?

. AIRCRAFT(#?)
(.)
```

```
AIRCRAFT(S)
COUNT =  1
FULL OUTPUT?
YES

-------------------------
MIG-25        FOXBAT-A


QRY>ITS SPEED?

. PERFORMANCE(?)(1!)[SPEED=*]
(.)

                                   SPEED
-------------------------------------------
MIG-25        FOXBAT-A           2.81    100000
                                 3.08     90000
                                 3.20     80000
                                 3.08     70000
                                 2.81     60000
                                 2.49     50000
                                 2.19     40000
                                 1.93     30000
                                 1.71     20000
                                 1.54     10000
                                 1.38         0


QRY>WHAT IS THE MINIMUM LENGTH OF INTERCEPTORS?

AIRCRAFT[ROLE={AW}I{R}]
. DIMENSION(?)(1!)[(MINIMUM) LENGTH=*]
(.)

ROLE                                 LENGTH
-------------------------------------------
MIN LENGTH=AWI      SU-15        FLAGON-E       68.00


QRY>OF BOMBERS?

. AIRCRAFT[ROLE={F/}B]
(.)

ROLE                                 LENGTH
-------------------------------------------
MIN LENGTH=F/B      MIG-17       FRESCO         36.33
```

A. A Session with the Demonstration System

```
QRY>PLEASE DESCRIBE CREWS.

AIRCRAFT CREW(?)
(.)

64 CREW
ONLY CREW SIZES ARE KNOWN.


QRY>HOW MANY AIRCRAFT CARRY 2 CREWMEN?

AIRCRAFT(*?)
. CREW[#=2]
(.)

AIRCRAFT(S)
COUNT =  4
FULL OUTPUT?
YES

-------------------------
MIG-25        FOXBAT-B
MIG-25        FOXBAT-A
MIG-25        FOXBAT
MIG-23U       FLOGGER-C


QRY>HOW MANY CARRY 1?

. AIRCRAFT(*?)
. CREW[#=1]
(.)

AIRCRAFT(S)
COUNT =  25
FULL OUTPUT?
NO


QRY>..

*** END AQT ***
```

References

[1] A. Aho and J. Ullman. Principles of Compiler Design. Reading,
MA: Addison-Wesley, 1978.

[2] E. Codd. A relational model of data for large shared data banks.
Comm. ACM 13 (June, 1970), pp. 377-387.

[3] N. Chomsky. Syntactic Structures. The Hague: Mouton, 1957.

[4] J.M. Foster. Automatic Syntactic Analysis. New York:
American Elsevier, 1970.

[5] L. Harris. User oriented data base query with the ROBOT natural
language query system. Int. J. Man-Machine Studies 9:6 (November, 1977),
697-713.

[6] G. Hendrix, E. Sacerdoti, D. Sagalowicz, and J. Slocum. Developing
a natural language interface to complex data. ACM Transactions on
Database Systems 3 (June, 1978), pp. 105-147.

[7] D. Knuth. Semantics of context-free languages. Math. Systems
Theory 2:2, 127-145.

[8] C. Mah. PARLEZ system description. PAR Report No. 79-8,
PAR Corp, Rome, NY, January 30, 1979.

[9] V. Pratt. A linguistics oriented programming language. A.I. Lab
Memo. No. 277, Artificial Intelligence Laboratory, Massachussets
Institute of Technology, February, 1973.

[10] F. Thompson and B. Thompson. Practical natural language processing:
the REL system as prototype. In Advances in Computers 13, M. Rubinoff
and M.C. Yovits, eds., New York: Academic Press, 1975.

[11] J. Tukey. Exploratory Data Analysis. Reading, MA: Addison-Wesley,
1977.

[12] D. Waltz. An English language question answering system for a large
relational data base. Comm. of the ACM 21 (July, 1978), pp. 526-539.

[13] W. S.Y. Wang, C.C. Liao, R. Gaskins, M.S. Wang, et. al.
Quince system; state-of-the-art-review. RADC-TR-78-153,
Rome Air Development Center, June, 1978, B029370L.

[14] T. Winograd. Understanding Natural Language. New York:
Academic Press, 1972.

[15] W. Woods. Transition network grammars for natural language
analysis. Comm. ACM 13:10 (October ,1970), 591-606.

[16] ___. RTSS advanced operating capabilities: Storage and
Retrieval Processor (SARP III). Technical Manual UM-RTSS-SARP-02,
Bunker-Ramo Corp, Westlake Village, CA, November 5, 1976.

APPENDIX B

Translation Tables for Soviet Aircraft Data

RELATION NAME TABLE

NAME	RECORD TYPE	ANCESTOR ID	LINKAGE
AIRCRAFT	1	0	1-1
ARMAMENT	3	1	1-1
CONFIGURATION	4	2	1-M
CREW	1	1	1-1
DIMENSION	1	1	1-1
DIMENSION	1	9	1-1
DIMENSION	1	15	1-1
ENGINE	1	1	1-1
FUSELAGE	1	1	1-1
IDENTIFICATION	1	1	1-1
IDENTIFICATION	1	8	1-1
PERFORMANCE	2	1	1-1
WEAPON	4	3	1-M
WEIGHT	1	1	1-1
WING	1	1	1-1

Translation Tables for Soviet Aircraft Data

FIELD CORRESPONDENCE TABLE

	RELN	REC TYPE	DATA TYPE	FIELD OFFSET	LENGTH
SERVICE NAME	1	1	A	0	12
NATO NAME	1	1	A	12	12
CREW	4	1	I	82	2
MANUFACTURER	1	1	A	88	12
ROLE	1	1	A	100	8
FIRST FLIGHT	1	1	A	108	10
WHEN IT[FIRST] ENTERED SERVICE	1	1	A	120	10
LENGTH	6	1	R	32	4
HEIGHT	6	1	R	36	4
CLASS	13	4	A	0	8
ACCOMMODATION	4	1	I	84	2
SPAN[SWEPT]	7	1	R	24	4
AFTERBURNER	8	1	A	68	1
POWER	8	1	I	64	2
MANUFACTURER	11	1	A	48	16
TYPE	11	1	A	44	4
COMBAT RADIUS	12	1	I	144	2
SPEED[VS ALTITUDE]	12	2	R	0	4
SIZE	13	4	I	48	2
EMPTY WEIGHT	14	1	I	72	4
GROSS WEIGHT	14	1	I	76	4
EXTENDED WEIGHT	14	1	I	80	4
EMPTY ..	14	1	I	72	4
GROSS ..	14	1	I	76	4
EXTENDED ..	14	1	I	80	4
TYPE	13	4	A	44	2
MFG DES	13	4	A	36	8
NATO NAME	13	4	A	8	12
ROCKET/POD	13	4	I	52	2
MFG	13	4	A	20	16
NAME	1	1	A	12	12
WEIGHT	14	1	I	76	4
..	14	1	I	76	4
#	13	4	I	4	2
#	8	1	I	40	2
#	4	1	I	84	2
LENGTH	5	1	R	32	4
HEIGHT	5	1	R	36	4
SERVICE ALTITUDE	12	1	I	140	4
ALTITUDE	12	2	I	-4	4
POWER RATING	11	1	I	64	2
COMBAT RADIUS	12	1	I	144	2

```
-----------------------------------------------------
* INTER-RELATIONAL RECORD LINKS
 TO    | FROM  | LINK LINK
 COORD | COORD | TYPE FIELD
 1   1   0   0     --    0  0      ; TOP-LEVEL INDICATOR
 1   4   1   1     --    0  0      ; AIRCRAFT TO CREW
 1   5   1   1     --    0  0      ; AIRCRAFT TO DIMENSION
 1   8   1   1     --    0  0      ; AIRCRAFT TO ENGINE
 1   9   1   1     --    0  0      ; AIRCRAFT TO FUSELAGE
 1  10   1   1     --    0  0      ; AIRCRAFT TO IDENTIFICATION
 1  14   1   1     --    0  0      ; AIRCRAFT TO WEIGHT
 1  15   1   1     --    0  0      ; AIRCRAFT TO WING
 1   6   1   9     --    0  0      ; FUSELAGE TO DIMENSION
 1   7   1  15     --    0  0      ; WING TO DIMENSION
 1  11   1   8     --    0  0      ; ENGINE TO IDENTIFICATION
 1  12   1   1     --    0  0      ; AIRCRAFT TO PERFORM.(RECTYP=1)
 2  12   1   1     ZZ  152  2      ; AIRCRAFT TO PERFORMANCE
 3  13   3   3     YY    0  2      ; CONFIGURATION TO WEAPON
 3   3   3   2     XX    4  2      ; ARMAMENT TO CONFIGURATION
 3   2   1   1     N-  148  2      ; AIRCRAFT TO ARMAMENT
 4   3   3 101     N-    0  2      ; CONFIGURATION ELEMENT TO WEAPON
 3 101   3 100     YY    0  2      ; CONFIGURATION TO ELEMENTS
 3 100   3   2     XX    4  2      ; ARMAMENT TO CONFIGURATION
* INTRA-RELATIONAL RECORD LINKS

 4  13   3   0     N-    0  2      ; WEAPON COUNT TO WEAPON

(COORDINATE PAIR = RECORD TYPE + RELATION)
-------------------------------------------------------
* RECORD TYPE ACCESS TABLE
ACCESS  SIZE  FILE NAME
TYPE
D        200   DK:AQT.DAT           ; MAIN AIRCRAFT RECORDS
D         50   DK:ALTSP.DAT         ; SPEED VS ALTITUDE TABLES
D         12   DK:ARMBAS.NDX        ; ARMAMENTS INDEX (WITH COUNTS)
D         64   DK:ARMBAS.DAT        ; ARMAMENTS RECORDS
```

Translation Tables for Soviet Aircraft Data

```
----------------------------------------------------------
* MANDATORY FIELDS TABLE FOR OUTPUT
 COORD   FIELD   DATA
         REQ'D   TYPE
 1   1    0 12    A      ; ALWAYS PRINT OUT SERVICE NAME
 1   1   12 12    A      ;     AND NATO NAME FOR AIRCRAFT
 4 13     8 12    A      ; IDENTIFY WEAPON BY NATO NAME
 4 13    44  2    A      ; WEAPON TYPE
 4 13     0  8    A      ; WEAPON CLASS
 2 12    -4  4    L      ; ALTITUDE WITH SPEED
 1   8   44  4    A      ; ENGINE TYPE
 1 11    44  4    A      ;   ( " )
 3 100   -1  6    A      ; CONFIGURATION MARKERS
 3 101    4  2    I      ; WEAPON COUNT (CONFIGURATION)
 4   3   44  2    A      ; WEAPON TYPE  (CONFIGURATION)
 4   3    0  8    A      ; WEAPON CLASS (CONFIGURATION)
 4   3    8 12    A      ; WEAPON NATO NAME

(COORDINATE PAIR = RECORD TYPE + RELATION)
----------------------------------------------------------
```

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of technical competence is provided to ESD Program Offices (POs) and other ESD elements. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.